

AD-A243 214



WL-TR-91-1060

PATTERN THEORY: AN ENGINEERING
PARADIGM FOR ALGORITHM DESIGN



T. ROSS
M. NOVISKEY
T. TAYLOR
D. GADD

Applications Branch
Mission Avionics Division

26 July 1991

Final Report for Period October 1988 - October 1990

DTIC
ELECTE
DEC 11 1991
S B D

Approved for public release; distribution unlimited

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

91-17379



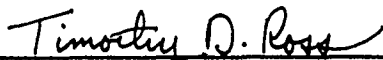
01 1000 000

NOTICE

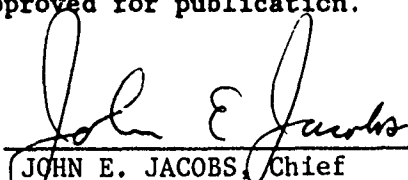
When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility nor any obligation whatsoever. The fact that the government may have formulated, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise in any manner construed, as licensing the holder or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

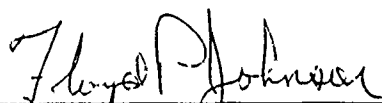


TIMOTHY D. ROSS
Project Engineer
System Concepts Group



JOHN E. JACOBS, Chief
System Concepts Group
Applications Branch

FOR THE COMMANDER



FLOYD P. JOHNSON, Chief
Applications Branch
Mission Avionics Division

If your address has changed, if you wish to be removed from our mailing list or if the addressee is no longer employed by your organization, please notify WL/AART-2, Wright-Patterson AFB, OH 45433-6542 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 26 07 91	3. REPORT TYPE AND DATES COVERED Final Oct 88 - Oct 90		
4. TITLE AND SUBTITLE Pattern Theory: An Engineering Paradigm for Algorithm Design		5. FUNDING NUMBERS WU 76290207 PE 62204F PR 7629 TA 02 WU 07		
6. AUTHOR(S) Timothy D. Ross, Michael J. Noviskey, Timothy N. Taylor, and David A. Gadd				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Avionics Directorate, WL, AFSC WL/AART-2 Wright-Patterson AFB OH 45433-6543 Timothy D. Ross, et. al. (513) 255-3215		8. PERFORMING ORGANIZATION REPORT NUMBER WL-TR-91-1060		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER		
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report proposes "Pattern Theory" as a basis for an engineering theory of algorithm design. Pattern Theory (PT) begins with a general statement of the problem and then makes deliberate specializations. The problem of finding a pattern in a function is the essence of algorithm design. The key to PT is its measure of pattern-ness: Decomposed Function Cardinality (DFC). Low DFC indicates pattern-ness. The principal result is a demonstration of the generality with which DFC measures pattern-ness. This generality is supported theoretically by relating DFC to time complexity, program length and circuit complexity. A test is developed, based on DFC, for whether or not a function will decompose. This test is used in Ada Function Decomposition (AFD) programs. AFD produces a decomposition (i.e. an algorithm in combinational form) and DFC. The generality of DFC is also supported experimentally. The Pattern Theory approach to machine learning and data compression demonstrated greater generality than other approaches. The DFC's of over 800 nonrandom functions (numeric, symbolic, string based, graph based, images and files) were measured. Roughly 98% of the nonrandom functions had low DFC versus less than 1% for random functions. AFD found the classical algorithms for several functions.				
14. SUBJECT TERMS Algorithm, Pattern Recognition, Function Decomposition, Machine Learning, Computational Complexity, Computers, Representation, Program Length, Extrapolation, Computing Theory.			15. NUMBER OF PAGES 213	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Acknowledgements

There were many people who contributed to this project. Prof. Alan Lair of the Air Force Institute of Technology and Messrs Devert Wicker and Steve Thomas of Wright Laboratory acted as consultants. During the summer of 1989 we had four temporary employees involved in the project. Mr. Mike Findler, (Arizona State University) worked here under the Air Force Office of Scientific Research (AFOSR) Graduate Summer Research Program (GSRP) [18]. Mr. Chris Vogt (Harvey Mudd College), Ms. Tina Normand (Miami University, Ohio) and Mr. John Langenderfer (Wright State University) were all summer hires. Mr. Vogt implemented the Ada code for the function decomposition algorithms and wrote the user's guide (Appendix B). Ms. Normand developed several of the combinatorics results of Section 5.2. Mr. Langenderfer performed the analysis of the relationship between the pattern-ness of a function and the pattern-ness of its inverse. The summer of 1990 brought even more help. There were three participants in the AFOSR Summer Faculty Research Program (SFRP). Prof. Mike Breen of Alfred University found and corrected several problems in the development of the Basic Decomposition Condition (see [8] and Section 5.2) and proved the set intersection size result of Section 6.6. Prof. Thomas Abraham of Saint Paul's College performed the Perceived Pattern-ness experiment (see [1] and Section 6.5). Prof. Thomas Gearhart of Capitol University worked mostly on the Convergence Method [25]; however, he performed most of the Neural Net experiments of Section 6.6 and made important contributions through participation in meetings and personal discussions. Ms. Shannon Spittler (Miami University, Ohio) contributed in several areas as a summer hire, especially in the generation and reduction of data. Messrs Mark Boeke and Michael Chabinyk were both made available by the AFOSR High School Apprenticeship Program. They, with Lt Taylor, developed the pattern phenomenology database software and produced many of the initial results [5, 11]. In addition to those who made direct technical contributions, several persons had important roles in the project. Mr. Leslie Lawrence of Wright Lab's Plans Office coordinated all of our AFOSR support. Ms. Peggy Saurez provided prompt and professional secretarial support whenever needed. Mr. John E. Jacobs was the immediate supervisor of all the authors and his guidance was essential to the project. Reviews by the next higher level of management, first Mr. Arther A. Duke and then Mr. F. Paul Johnson, kept the project on track. Support from the next higher level of management, first Mr. Edward Deal and then Mr. Les McFawn, was essential in that they allocated the needed resources for the project.



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification _____	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Avail and/or Special

Contents

1	Introduction	1
2	Background	5
2.1	Pattern Theory	5
2.1.1	Introduction to Pattern Theory	5
2.1.2	What is a Pattern?	5
2.2	Background of Related Disciplines	12
2.2.1	Recognizing Patterns — the Many Disciplines	12
2.2.2	Pattern Recognition	12
2.2.3	Artificial Intelligence	13
2.2.4	Algorithm Design	15
2.2.5	Computability	15
2.2.6	Computational Complexity	16
2.2.7	Data Compression	16
2.2.8	Cryptography	17
2.2.9	Switching Theory	17
2.2.10	Summary	17
3	The Pattern Theory Paradigm	19
3.1	Why is Pattern Theory Needed?	19
3.1.1	Offensive Avionics as a Potential Application	19
3.1.2	Importance of Computing Power in Offensive Avionics	19
3.1.3	Importance of Algorithms in Computing Power	20
3.1.4	Role of a "Design Theory"	22
3.1.5	The Need for a Design Theory for Algorithms	23
3.1.6	Summary	24
3.2	The Pattern Theory Approach	24
3.2.1	The "Given and Find" Characterization of a Design Theory	24
3.2.2	Definition, Analysis and Specialization	25
3.3	The General Problem of Computational System Design	26
3.3.1	Computation and Functions	26
3.3.2	Representation	27
3.3.3	Figures-of-Merit	28
3.3.4	Problem Statement	29

3.4	The Pattern Theory 1 Problem as a Special Problem in Computational System Design	30
3.4.1	Special Rather Than General Purpose Computers	31
3.4.2	Single Function Realization	31
3.4.3	Input Representation System	31
3.4.4	Output Representation System	32
3.4.5	Functions	32
3.4.6	Definition versus Realization	33
3.4.7	Figure-of-Merit for the PT 1 Problem	33
3.4.8	Kinds of Patterns	34
3.4.9	Problem Statement	34
3.5	Summary	34
4	Decomposed Function Cardinality as a Measure of Pattern-ness	37
4.1	Introduction	37
4.2	Decomposed Function Cardinality	47
4.3	Decompositions Encoded as Programs	48
4.3.1	Introduction	48
4.3.2	Encoding Procedure	49
4.3.3	Length of an Encoding	50
4.3.4	\mathcal{R}' Includes All Optimal Representations	51
4.3.5	Properties of Encodings	52
4.3.6	Decomposed Function Cardinality and Program Length	53
4.4	Decomposed Function Cardinality and Time Complexity	55
4.5	Decomposed Function Cardinality and Circuit Complexity	58
4.6	Summary	59
5	Function Decomposition	61
5.1	Introduction	61
5.2	The Basic Decomposition Condition	62
5.2.1	Introduction	62
5.2.2	An Intuitive Introduction to the Decomposition Condition	62
5.2.3	The Formal Basic Decomposition Condition	69
5.2.4	Non-Trivial Basic Decompositions	73
5.2.5	Negative Basic Decompositions	76
5.3	The Ada Function Decomposition Programs	79
5.3.1	Program Functional Description	80
5.3.2	Program Software Description	82
5.3.3	Versions of the AFD Algorithm	82
5.4	Ada Function Decomposition Program Performance	88
5.4.1	Cost Reduction Performance	89
5.4.2	Run-Time Performance	93
5.4.3	Summary	99
5.5	Summary	99

6	Pattern Phenomenology	101
6.1	Introduction	101
6.2	Randomly Generated Functions	102
6.2.1	Introduction	102
6.2.2	Completely Random Functions	103
6.2.3	Functions with a Specific Number of Minority Elements	108
6.2.4	Functions with a Specific Number of Don't Cares	110
6.3	Non-randomly Generated Functions	113
6.3.1	Numerical Functions and Sequences	114
6.3.2	Language Acceptors	122
6.3.3	String Manipulation Functions	123
6.3.4	A Graph Theoretic Function	127
6.3.5	Images as Functions	129
6.3.6	Data as Functions	130
6.3.7	Summary	136
6.4	Patterns as Perceived by People	139
6.4.1	Effect of the Order of Variables on the Pattern-ness of Images	139
6.5	Pattern-ness Relationships for Related Functions	142
6.5.1	Functions and Their Complements	142
6.5.2	Functions and Their Inverses	143
6.6	Extrapolative Properties of Function Decomposition	145
6.6.1	Introduction	145
6.6.2	FERD Experiments	146
6.6.3	FERD and Neural Net Comparisons	154
6.6.4	FERD Theory	156
6.6.5	Summary	162
6.7	Summary	162
7	Conclusions and Recommendations	163
8	Summary	165
A	Program Length and the Combinatorial Implications for Computing	167
A.1	Mathematical Preliminaries	167
A.1.1	Basic Definitions	167
A.1.2	Combinatorics	168
A.2	Program Length Constraints for Computation	171
A.2.1	Introduction	171
A.2.2	Programmable Machines	172
A.2.3	Maximum-Minimum Program Length for Finite and Transfinite Sets	176
A.2.4	Average-Minimum Program Length Bound for Finite Sets	183
A.3	Summary	191

List of Figures

2.1	The Algorithm Design Process	6
2.2	The Grand Scheme for Algorithm Design	7
2.3	Pattern Theory Phase 1	7
2.4	Pattern Theory Phase 2	8
2.5	Pattern Theory Phase 3	8
2.6	Pattern Theory Phase 4	9
2.7	Patterned and Un-Patterned Objects	11
2.8	Neural Net Paradigm	14
2.9	Model-Based Reasoning Paradigm	16
3.1	Time Complexity of Algorithms	21
3.2	Typical Algorithm Input Sizes	22
4.1	f as a Composition of Smaller Functions	39
4.2	Decomposition of Addition	40
4.3	Decomposition of a Palindrome Acceptor	41
4.4	Decomposition of a Prime Number Acceptor	43
4.5	Decomposition of an Image	44
4.6	An Image of "R"	44
4.7	Similar Decompositions, One Recursive, One Not	57
5.1	Form of a Decomposition	68
5.2	Form of a More General Decomposition	68
5.3	Example Decomposition	70
5.4	Relationship Between ν and $[V_1]$, where $D(f) = [f]$	75
5.5	The Basic Decomposition	75
5.6	DECOMP.RECORD Data Structure	80
5.7	FIND_LOWEST_COST Flow Chart	81
5.8	FIND_LOWEST_COST Psuedo-Code	82
5.9	Algorithm Stages	83
5.10	Compilation Dependencies	84
5.11	NU_MAX for Each Version of the AFD Algorithm	86
5.12	Neural Net Gross Architecture	91
5.13	Detailed Architecture of a Neural Net Component	92
5.14	Specific NN Architectures	92

5.15	Run-time versus DFC for Functions on Eight Variables	97
5.16	Run-Time versus Number of Minority Elements for Six Variable Functions	97
5.17	Run-Time versus Number of Minority Elements for Seven Variable Functions	98
5.18	Run-Time versus Number of Minority Elements for Eight Variable Functions	98
6.1	Number of Functions versus DFC for n up to 24	106
6.2	Number of Functions versus DFC for $n = 5$	107
6.3	DFC With Respect to Number of Minority Elements, $n=4$	110
6.4	DFC With Respect to Number of Minority Elements, $n=5$	111
6.5	DFC With Respect to Number of Minority Elements, $n=6$	111
6.6	DFC With Respect to Number of Minority Elements, $n=7$	112
6.7	DFC With Respect to Number of Minority Elements, $n=8$	112
6.8	DFC as a Function of the Number of Cares	113
6.9	DFC as a Function of the Number of Cares, $n = 7$	114
6.10	Font 0 Images and DFC	131
6.11	Font 1 Images and DFC	131
6.12	Font 2 Images and DFC	132
6.13	Font 3 Images and DFC	132
6.14	Font 4 Images and DFC	132
6.15	Variable Permutations for Characters 177 and 197 of Font 0	140
6.16	Variable Permutations for Characters 15 and 1 of Font 0	140
6.17	Variable Permutations for Characters 10 of Font 0 and 48 of Font 2	141
6.18	Variable Permutations for Characters 51 of Font 2 and 31 of Font 3	141
6.19	Relationship Between Functions of a Given DFC and the Average DFC of Their Inverses	144
6.20	Learning Curve for XOR Function	147
6.21	Learning Curve for Parity Function	147
6.22	Learning Curve for Majority Gate Function	147
6.23	Learning Curve for a Random Function with Four Minority Elements	148
6.24	Learning Curve for the Symmetric Function	148
6.25	Learning Curve for Primality Test on Seven Variables	148
6.26	Learning Curve for Primality Test on Nine Variables	149
6.27	Learning Curve for a Random Function	149
6.28	Learning Curve for Font 1 "P"	149
6.29	Learning Curve for Font 1 "T"	150
6.30	Learning Curve for Font 0 "R"	150
6.31	Learning Examples for the Parity Function	151
6.32	Learning Examples for the Letter "R"	152
6.33	The Pattern Theory Logo	152
6.34	Learning Curves for Random Functions on 4 Through 10 Variables	153
6.35	Number of Samples Required for ≤ 10 errors	153

6.36	Neural Net Learning Curve for XOR Function	154
6.37	Neural Net Learning Curve for Parity Function	154
6.38	Neural Net Learning Curve for Majority Gate Function	155
6.39	Neural Net Learning Curve for the Symmetric Function	155
6.40	Second Neural Net's Learning Curve for the Step Function	157
6.41	Second Neural Net's Learning Curve for the Majority Gate Function .	157
.		
A.1	A Machine's Interfaces	172
A.2	"Programs" in a Communications Context	173
A.3	Simplified RAM Model	174
A.4	BASIC Allows for Tabular Data Structures	187
A.5	An Example Table Machine	190

List of Tables

1.1	Find an Algorithm for This Function	2
1.2	Find an Algorithm for This Function	2
2.1	Recognizing a Pattern in a Function	9
2.2	Recognizing a Pattern in a Function	10
4.1	Function Cardinality of h is 8	38
4.2	The Function Cardinality of f and g is 16	38
4.3	Functions that Compose f	39
4.4	Addition on Six Variables (Four Output Functions)	40
4.5	Addition Components a_1 and c_1 (XOR and AND)	40
4.6	Addition Components a_2 and c_2	41
4.7	Palindrome Acceptor on Six Variables	42
4.8	Palindrome Acceptor Component a_1 (NOT XOR)	42
4.9	Palindrome Acceptor Component b (AND)	42
4.10	Prime Number Acceptor Components a_1 and a_2	43
4.11	Letter R Components c and d	43
4.12	Letter R Component a	45
4.13	Letter R Component b	45
5.1	A Table Representation of a Function	63
5.2	A 2-D Table of a Function With Respect to a Partition of its Variables	64
5.3	A Second 2-D Table of a Function With Respect to a Partition of its Variables	64
5.4	A Table Representation of a Function	64
5.5	A 2-D Table of a Function With Respect to a Partition of its Variables	65
5.6	A Table Representation of Function g	66
5.7	A Partition Matrix (2-D Table) of g	66
5.8	A Partition Matrix of g With ϕ Defined	66
5.9	g Defined by G and ϕ	67
5.10	Various Forms of Z	68
5.11	Partition Matrices	69
5.12	Functions f and g	76
5.13	AFD Algorithm Version Space	88
5.14	Average DFC for Set A and Set B	90

5.15 DFC of NN Like Architectures	91
5.16 AFD-DFC of NN Like Architectures	93
5.17 Average Run-time for Set A and Set B	94
5.18 Run-times for Functions That Did Not Decompose	95
5.19 Run-Times for Functions That Did Decompose	96
6.1 Number of Functions for a Given DFC	105
6.2 Number of Minority Elements Required for a Given Cost	111
6.3 Addition.	115
6.4 Subtraction.	116
6.5 Multiplication.	116
6.6 Modulus.	117
6.7 Remainder.	117
6.8 Square Root.	118
6.9 Cube Root.	118
6.10 Sine.	119
6.11 Logarithm.	119
6.12 Miscellaneous Numerical Functions.	119
6.13 Primality Tests.	120
6.14 Fibonacci Numbers.	120
6.15 DFC of Lucas Functions.	121
6.16 DFC of Binomial Coefficient Based Functions.	121
6.17 DFC of Greatest Common Divisor Function.	122
6.18 DFC of the Determinant Function.	122
6.19 Sample Languages.	124
6.20 DFC of Language Acceptors.	125
6.21 Miscellaneous String Manipulation Functions.	125
6.22 Sorting Eight 1-Bit Numbers.	126
6.23 DFC of Sorting Four 2-Bit Numbers.	126
6.24 Input Bits Represent Arcs	127
6.25 Additional Input Bits for Arcs to Self	128
6.26 DFC of the Various k-clique Functions on a Graph With 5 Nodes . .	128
6.27 DFC of the Various k-clique Functions on a Graph With Four Nodes	129
6.28 Turbo Pascal V5.5 Font Sets	129
6.29 Character Images DFC Statistics	130
6.30 DFC and Data Compression Results for Typical Files	134
6.31 Data Compression Summary for Typical Files	134
6.32 Data Compression Summary for Atypical Files	135
6.33 Decomposition Summary for Non-Randomly Generated Functions . .	137
6.34 Larger n Shows Greater Decomposability	138
6.35 Character Images	140
6.36 Permutations of Variables	141
6.37 Number of Functions and Inverses with a Given Cost Combination . .	144
6.38 FERD (F) and NN (N) Error Comparison	156

A.1 Fraction of Functions Computable by NN	189
--	-----

Chapter 1

Introduction

Can you invent an algorithm for the function defined in Table 1.1? That is, can you write a computer program that generates $f(x)$ when given x and not use a brute force table look-up? What about the function in Table 1.2?¹ Think about how you invented these algorithms. Were your algorithms based on a *pattern* in the function? For example, did you notice that for the first example the output is 1 if and only if the input, taken as a string, is symmetric about its center? What do you think about computers finding patterns like these? It seems that some people are surprised that computers cannot already do this. If we know ahead of time that the function has some specific structure then we can write a program to fine tune that structure; but, we do not have computers that can find basic structures in a very general setting. Others are surprised that someone would even suggest that computers might be able to do this. The invention of algorithms has been equated with scientific discovery (e.g. [32] which makes one balk at the idea of automating algorithm design. We believe that algorithm design is at most a subset of scientific discovery and that it is a subset that can be automated. Further, we believe that the first step towards automation is to develop a solid theoretical understanding of this pattern finding ability that characterizes algorithm design. This theoretical understanding must in turn be built on a solid understanding of "pattern."

The algorithms in use today were invented by people. There are other similar engineering products, such as estimation systems, control systems, communication systems, that were designed by people, but with a fundamentally different dependence upon the cleverness of the designers. That is, in the traditional engineering problems, there is an engineering theory that guides the designer. People must invent new algorithms without the aid of an engineering theory. The difference between the algorithm engineering problem and many other engineering problems is reflected in the difference between "invent" and "design." Webster [66] defines "invent:"

"...to produce ...through the use of the imagination or of ingenious thinking ..."

¹One algorithm is to treat the first 2 bits as one number and the second 2 bits as a second number and then $f(x)$ is the arithmetic sum of these two numbers.

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 1.1: Find an Algorithm for This Function

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
0	0	0	0	000
0	0	0	1	001
0	0	1	0	010
0	0	1	1	011
0	1	0	0	001
0	1	0	1	010
0	1	1	0	011
0	1	1	1	100
1	0	0	0	010
1	0	0	1	011
1	0	1	0	100
1	0	1	1	101
1	1	0	0	011
1	1	0	1	100
1	1	1	0	101
1	1	1	1	110

Table 1.2: Find an Algorithm for This Function

and "design:"

"...to create, fashion, execute, or construct according to plan ..."

It seems that algorithms are invented while estimation systems, control systems, etc. are designed. We believe that the difference between invention and design is simply the existence of an engineering theory. We need an engineering theory to allow algorithm *design*.

This report introduces "Pattern Theory." Pattern Theory consists of a formal definition of pattern (or structure), an approach to finding the pattern when it exists, and a characterization of various phenomena with respect to this structure. The principal objective of this report is to demonstrate that many kinds of practically important patterns are well reflected in this formal definition.

Chapter 2 describes the need for an engineering theory of algorithm design. Chapter 3 describes Pattern Theory which is our approach to a design theory for algorithms. The key to our approach is a measure of algorithm good-ness that we call Decomposed Function Cardinality (DFC). Chapter 4 defines this measure and relates it to the more conventional measures. Function Decomposition is the method for optimizing with respect to DFC. Chapter 5 develops the theory behind function decomposition and describes computer programs for accomplishing decompositions. We equate the existence of a good algorithm for a given function and the existence of a "pattern" in that function. So the design of a good algorithm is the same as finding the pattern in a function and we think of DFC as a measure of the pattern-ness of a function. Chapter 6 reports on the results of applying this measure to a variety of functions; we call this class of results "Pattern Phenomenology."

Chapter 2

Background

2.1 Pattern Theory

2.1.1 Introduction to Pattern Theory

The development of Pattern Theory began around 1986 at the Air Force Institute of Technology (AFIT), Wright Patterson Air Force Base, Ohio. One of this report's authors, then on Long-Term Full-Time training, Prof Alan V. Lair, of AFIT's Mathematics Department and Prof Matthew Kabrisky, of AFIT's Electrical Engineering Department, all played major roles in this early work. A discussion of many of the ideas that went into Pattern Theory was published in [48, 50, 51]. The name "Pattern Theory" was adopted after the International Conference on Pattern Recognition in 1988. Our paper at that conference was in a session entitled "Fuzzy Sets and Pattern Theory." All the other papers were clearly about Fuzzy Sets, so we must have been the Pattern Theory. A team of AART and visiting engineers continued the Pattern Theory work in the in-house Pattern Based Machine Learning (PBML) project whose results are the subject of this report. The PBML Project is generally referred to as Pattern Theory 1 (PT 1) in this report.

2.1.2 What is a Pattern?

An Introduction to the Pattern Theory Paradigm

It will be useful to briefly introduce the Pattern Theory (PT) paradigm to motivate the background. Chapter 3 is a detailed introduction to the PT paradigm. The basic problem is how do you go from a definition of a function to a computer realization of that function. The problem has some definition of a function as its starting point and a computer algorithm as its solution.

We divide the kinds of information that might constitute the definition into two classes: samples of the function and "other" information about the function. Figure 2.1 represents the algorithm design problem. The grand scheme of Pattern Theory is to eventually complicate this flow chart slightly by allowing "learned" algorithms

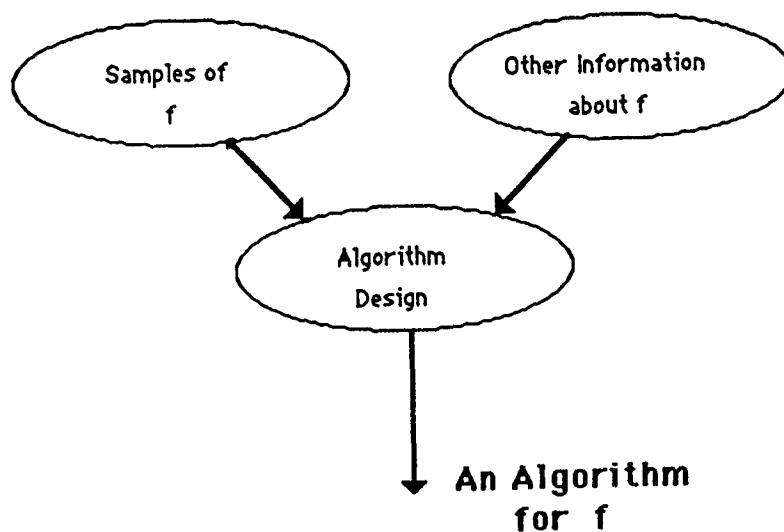


Figure 2.1: The Algorithm Design Process

to be added to the “other” information. By closing the loop we create an iterative approach to realizing more and more complicated functions. Figure 2.2 represents the iterative approach to algorithm design. This representation will be useful in explaining the phases of Pattern Theory and its relationship to other paradigms. Figures 2.3 through 2.6 represent the four planned phases of Pattern Theory. Pattern Theory Phase 1 concerns algorithm design by function decomposition when the function is defined by an exhaustive table. Pattern Theory Phase 2 concerns algorithm design by function decomposition when the function is defined by a combination of samples and limited other information. Initially the other information will simply be that the function has limited computational complexity. Pattern Theory Phase 3 concerns algorithm design by function decomposition when the function is defined by a combination of limited samples and robust other information. Pattern Theory Phase 4 concerns iteratively designing increasingly complex algorithms by function decomposition. This report is concerned with the results of the first phase. The second phase (PT 2) began as this report was being finished.

While there is no general theory for working the problem of algorithm design it has been recognized that finding some pattern in the function could be important (e.g. “Perhaps the most valuable concept of all in the invention of algorithms is that of recognizing patterns ...” [38] or “...many of the central problems of behavior, intelligence, and information processing are problems that involve patterns.” [62]). Pattern Theory is an attempt to formalize this pattern finding problem within the context of algorithm design. By a “pattern” we mean the structure, order or regularity in a function. Most people would have no trouble recognizing the patterns in the

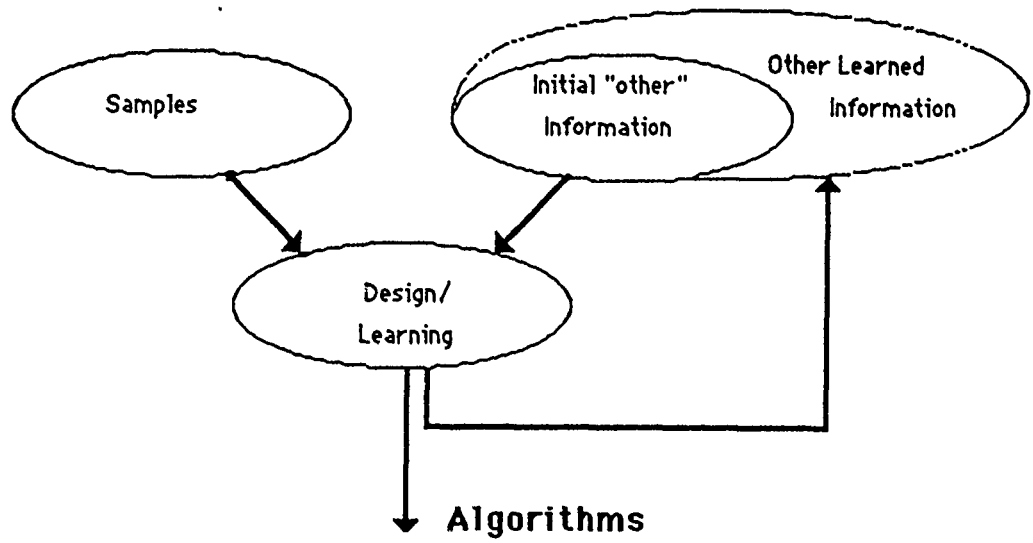


Figure 2.2: The Grand Scheme for Algorithm Design

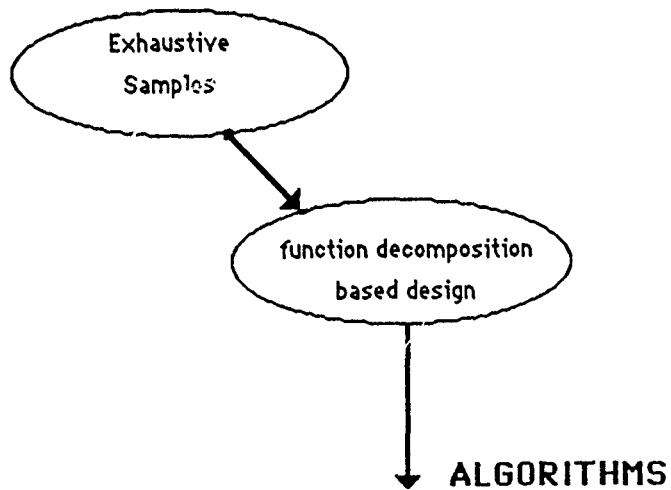


Figure 2.3: Pattern Theory Phase 1

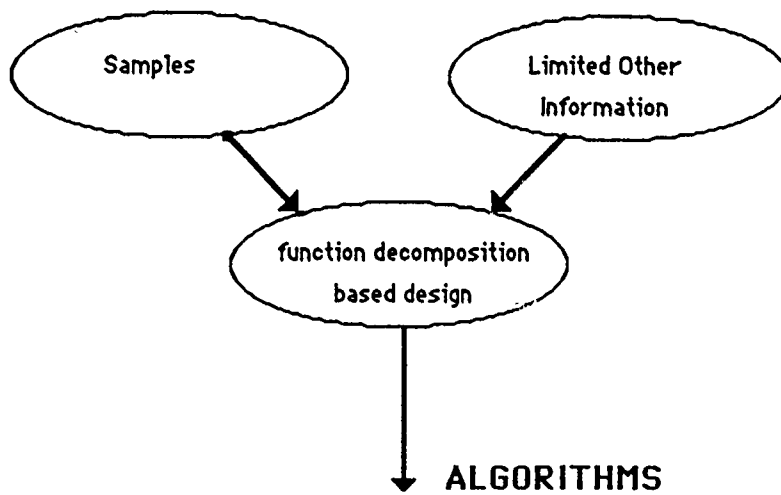


Figure 2.4: Pattern Theory Phase 2

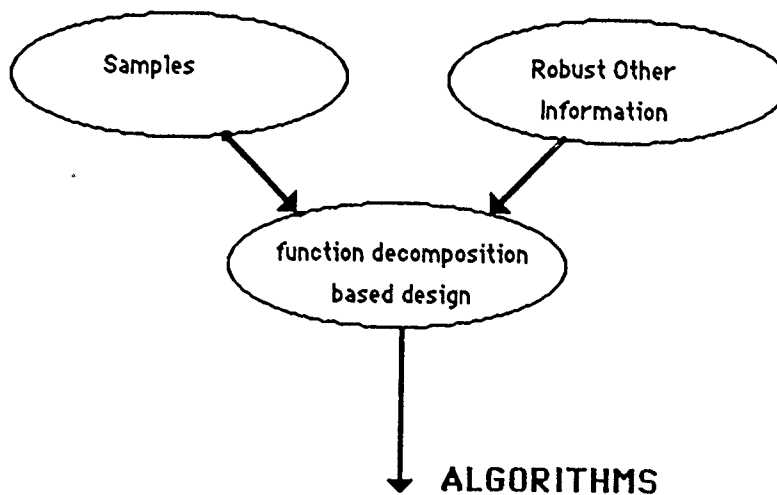


Figure 2.5: Pattern Theory Phase 3

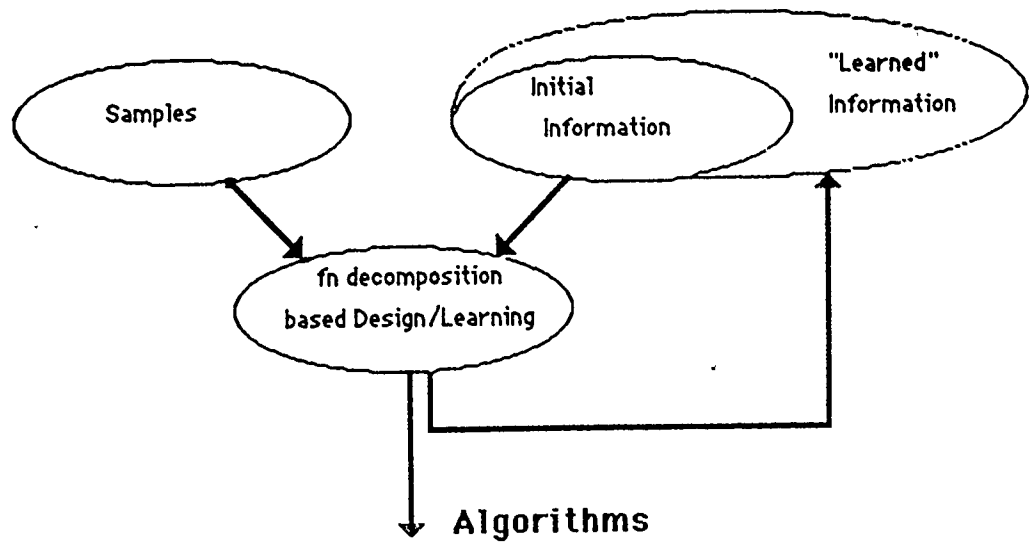


Figure 2.6: Pattern Theory Phase 4

x	$f(x)$
1	1
2	4
3	9
4	16
5	25
6	36
\vdots	\vdots

Table 2.1: Recognizing a Pattern in a Function

functions defined by Table 2.1¹ and Table 2.2². Pattern Theory concerns the problem of recognizing the patterns in functions that will allow their economical computation. But, what is a pattern?

Intuitive Ideas about Patterns

We are concerned with patterns in the sense of regularity, order, structure or the opposite of chaos. People seem to have a common sense notion of pattern-ness. This common sense notion of a pattern is supported by people's willingness to assign a pattern-ness ranking in experiments like Garner's [24] and those of Section 6.4. Patterns can occur in many different forms. Figure 2.7 has examples of patterned

¹ $f(x) = x^2$.

²Primality test.

x	$f(x)$
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	0
10	0
11	1
12	0
\vdots	\vdots

Table 2.2: Recognizing a Pattern in a Function

and unpatterned images, strings of letters, and sequences of numbers. Again on an intuitive level, patterns are easier to remember; for example, the sequence

17761812186519151941

is easier to remember (if you recognize the pattern) than a sequence like

73217519816234218192.

Patterns also seem to be easier to extrapolate; for example, we would have more confidence in guessing the next number in the sequence 2, 4, 6, 8, 10, 12, ... than in the sequence 5, 2, 7, 3, 5, 12, ...

Traditional Ideas about Patterns

Although there seems to be this common sense notion of pattern-ness, there has been little success in capturing this notion as a formal mathematical concept. References [48, 51] describe our assessment of the traditional formulations of pattern-ness.

Patterns and Simplicity

We feel that the most useful direction for exploring pattern-ness is the one which relates pattern-ness and simplicity of description. Simplicity is the opposite of complexity and computational complexity has a well developed theory. Therefore, through this connection to complexity, pattern-ness immediately has a rich theory.

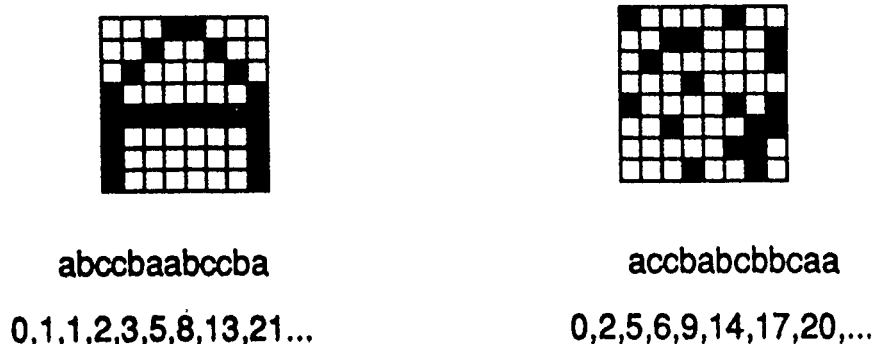


Figure 2.7: Patterned and Un-Patterned Objects

The Relativity Problem

A problem arises though because the theory is almost too “rich.” That is, there are many measures of complexity and pattern-ness is then *relative* to the measure used. Pattern Theory addresses this relativity problem by proposing that there is a special model of a computer and a measure that reflects the essence of complexity in the sense of patterns.

In a sense we have gone full circle. We started with the problem of finding economical representations of a function (i.e. an algorithm). We decided that recognizing patterns is important in this endeavor. Now we are saying that recognizing patterns is essentially the same as finding economical representations. Why even bring up the concept of patterns? The answer lies in the need for a concept of general computational complexity that does not currently have a name. This needed concept closely reflects the intuitive notion of a pattern so that is what we call it. We also like the connection this gives the problem to the early pattern recognition work. This early work in pattern recognition formed the basis for many current artificial intelligence problems. When you consider the problem of algorithm design as simply one of minimizing computational complexity, the temptation to choose a specific non-general measure of complexity is too strong. We lose sight of the idea of finding the basic structure (i.e. pattern) in the function. As a practical matter, we could develop all the “Pattern Theory” concepts in terms of traditional computational complexity. However, by talking about patterns we feel we more easily focus on the general or abstract complexity which is so important and it ties us into disciplines which we think are quite relevant.

2.2 Background of Related Disciplines

2.2.1 Recognizing Patterns — the Many Disciplines

In the following we will survey the disciplines relevant to Pattern Theory. Perhaps the most obvious discipline is pattern recognition (e.g. [16, 22, 29]). However, the modern approaches to pattern recognition do not treat patterns in our special sense. This position is developed in [48, 51]. Early pattern recognition research was concerned with special patterns, as are elements of modern research (e.g. [58, 63]). At one time, Pattern Recognition (PR) and Artificial Intelligence (AI) research had a great deal in common. This common philosophical base is quite relevant to Pattern Theory. However, the specific disciplines within PR and AI (e.g. statistical pattern recognition, syntactic pattern recognition, expert systems, neural nets) seem to have all diverged from the core problem. In all these disciplines, the basic structure of the problem must be recognized by the designer without theoretical tools or automation. Only after this basic structure is defined can theoretical tools or anything approaching automation be applied. Data Compression (e.g. [27]) can be considered as a problem of finding and exploiting patterns in data. This has an obvious connection with our problem. Within the data compression discipline the patterns are recognized by the designer of the data compression routine, again without theoretical tools or automation. As we have already mentioned, the complexity and computability disciplines of theoretical computer science are most related to Pattern Theory. We will make extensive use of computational complexity results. We will also show that computability is a sub-problem of complexity and of no special interest within our context (see Appendix A). Finally, the problem of designing electronic circuits (switching theory) is connected to Pattern Theory. We will see that with respect to our generalized measure of complexity, designing efficient circuits and designing efficient algorithms are the same problem. As you would expect, both problems depend on finding some pattern in the function to be realized. We make extensive use of function decomposition theory which was originally developed within the switching theory context.

2.2.2 Pattern Recognition

The relationship between the traditional field of pattern recognition and Pattern Theory is discussed in depth in [48, 49]. The following is a brief summary of that discussion.

The subject of pattern recognition can be divided up many ways. The most common is to consider the fields of statistical (also decision-theoretic, geometric or vector space) pattern recognition, syntactic (also structural or linguistic) pattern recognition and fuzzy methods of pattern recognition. The references [48, 49] use a slightly different division, emphasizing the role of a priori structure in designing recognizers. The a priori structure is the representation system or language used to express the recognition algorithm. Pattern Theory is an attempt to generalize this idea of a priori structure. Therefore, the role of a priori structure within traditional

pattern recognition is especially relevant. Most traditional pattern recognition is based on either a geometric or a syntactic structure. Reference [48] discusses the background of traditional pattern recognition in terms of these two structures.

The basic disconnect between Pattern Recognition and Pattern Theory lies in our belief that the interesting pattern finding phenomenon occurs in the *design* of recognition systems rather than in their operation. Reference [51] explains this position. This difference in perspective is reflected in the different approaches to research. In Pattern Recognition it is generally believed that a researcher should choose a single realistic problem (typically speech or character recognition). The PT approach is to study many simple problems (e.g. Chapter 6 reports on over 1000 different functions). The concern is that when we study only a single function, the researcher ends up doing the pattern finding and the so-called "pattern recognition" algorithm is simply a realization of the patterns recognized by the researcher. Studying many different kinds of functions makes it more difficult for the researcher to insert (deliberately or unconsciously) any humanly recognized patterns. This forces the machine to do some true pattern finding.

2.2.3 Artificial Intelligence

Machine learning, a problem of artificial intelligence (AI), might be thought of as an attempt to automate the process that we seek to understand. That is, we want to understand the process of defining an algorithm while machine learning seeks to automatically generate an algorithm. Therefore, Pattern Theory has a strong connection to machine learning.

We think of the artificial intelligence approach to this problem as one of figuring out how people do it and then attempting to model that process on a computer. For example, expert systems derive from the cognitive psychology model of thought and neural nets derive from the physiological model of the hardware involved in thought. It is possible that AI will come up with useful systems based on this approach without any understanding of the process at an abstract level. An often used analogy for AI is the problem of manned flight. In this analogy the AI approach would be analogous to the artificial bird approach. That is, we could design machines with bird-like properties since a bird is an existing system which performs the desired function. We are trying to take what might be called the "Wright" approach. That is we seek to understand the basic phenomenon that will allow us to design from first principles. This approach will not immediately lead to systems with practical value; however, we believe it is the only approach to continuing long term improvements.

In AI based machine learning,

"The human engineer specifies a weak method for a problem's solution that is (semi) automatically (...) streamlined by the system with experience." ³

³From Doug Fisher's Tutorial: Machine Learning and its Applications, July 1990.

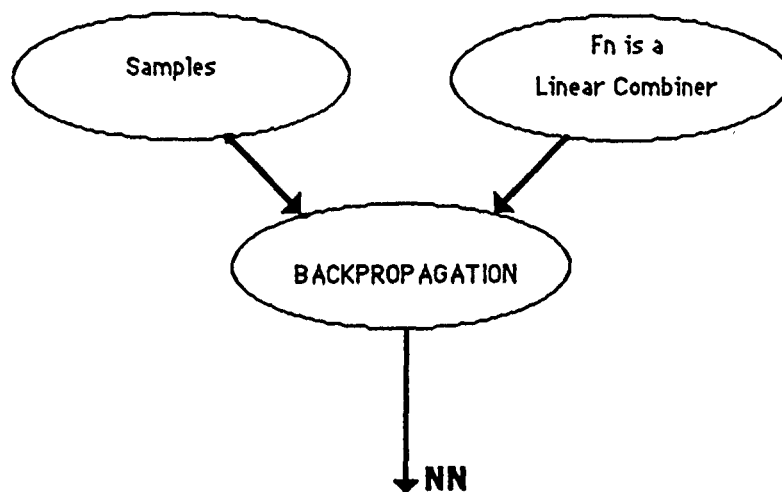


Figure 2.8: Neural Net Paradigm

We feel that the so called "weak method" constitutes a large fraction of the overall solution. The problem addressed in Pattern Theory includes the development of a weak method as well as the "automatic streamlining."

There are many approaches to machine learning. Learning within the context of expert systems include rule learning (e.g. [13]), adaptive figure-of-merits to improve non-exhaustive searches (e.g. [13]), and genetic algorithms (e.g. [15]). Some neural nets learn [53]. Within the discipline of pattern recognition there are learning methods for both geometric (e.g. [16]) and syntactic (e.g. [22]) systems. Adaptive systems (e.g. [37]) as used in estimation and control theory for non-linear systems have as many learning characteristics as AI systems.

We can characterize machine learning systems using the diagram in Figure 2.1. The "other" information includes an assumption that the desired function has a realization of the form used by the learning system. Take Neural Nets for example (Figure 2.8), the "other information" is an assumption that the desired function may be represented by the chosen architecture of thresholded linear combinations. Appendix A demonstrates that this assumption is surprisingly restrictive. The design approach then is back-propagation or some other method of assigning weights. The traditional machine learning paradigms are built around a specific structure. The key idea of Pattern Theory is that we want to find the structure that already exists in the function. We do not want to try to force fit a function to some structure that we chose ahead of time.

One AI approach, known as Abduction [44], uses the "chunking" idea of Miller [43]. The function decomposition approach of Pattern Theory also exhibits this chunking idea.

2.2.4 Algorithm Design

The texts on algorithm design (e.g. [3]) are quite different from the texts on most other electrical engineering design problems (e.g. circuit design, control system design, communications system design). Most electrical engineering design texts tell, in an almost cookbook fashion, how to solve problems of a given type. Typically you begin by developing a dynamic model of the system involved. Next, you apply some very general principles, such as modulation in communications or feedback in controls. Then there are some mathematically rigorous tools for optimizing the design. Finally there are methods for predicting performance and evaluating the design. By contrast, texts on algorithm design give a list of specific algorithms that you are to mix and match to your problem. They do not tell you how to come up with a new algorithm. If controls texts were like algorithm design texts they might give a table of feedback gains for specific plants and specific desired step responses, but they would not give the general relationship between feedback gain and system performance that control theory actually provides. It seems that if an engineer with a good understanding of control theory were to compete in solving a new controls problem with an engineer with no controls background, the engineer with knowledge of control theory would arrive at a much better design. However, if two engineers were to compete at discovering a new algorithm, the engineer with a background in algorithm design would seem to have little advantage (unless, of course, some previously discovered algorithm happened to fit the new problem). In summary, although you can find texts on algorithm design, they do not address design of fundamentally new algorithms.

In the introduction of an algorithm design text they may mention a general principle of algorithm design known as "divide and conquer," e.g. [7, p.3]. The function decomposition approach of Pattern Theory can be thought of as a formalization of the divide and conquer principle.

An important technology that is being developed and used in the Avionics Directorate is Model-Based Reasoning, especially its application to target recognition. From a Pattern Theory perspective, Model-Based Reasoning is not too different from traditional algorithm design. Referring again to Figure 2.1, model-based simply means that the "other" information is a collection of models. The algorithm design problem is classical; that is, we are left to our own inventiveness to turn the models into an algorithm (Figure 2.9).

2.2.5 Computability

The problem of computability would seem to be quite relevant to Pattern Theory. But it is not. Computability, in its formal sense, is tied to recursion.

"... because all evidence indicates that the class of partial recursive functions is exactly the class of effectively computable functions; ..." [35]

It seems clear that recursion is a desirable property in a function, but it is neither necessary nor sufficient for a function to be patterned. We say this because all functions

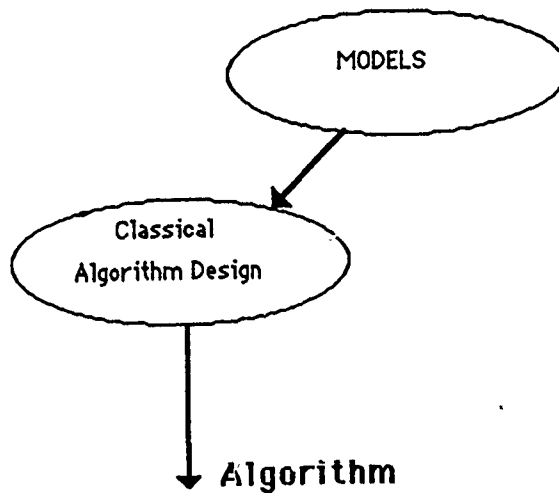


Figure 2.9: Model-Based Reasoning Paradigm

of interest in practical computing are finite; all finite functions are partial recursive; yet finite functions are not practically computable with high probability. There are of course many infinite functions (especially those on the real or natural numbers) that are of interest, but we never really try to compute them. We would always be satisfied with the ability to compute these function on some finite sub-domain. Therefore, the use of (and complete dependence on) infinite functions for interesting results in computability makes it of no practical use in Pattern Theory. We will argue later that recursion is of secondary importance in the general complexity used in Pattern Theory. Appendix A develops some classical computability results from a Pattern Theory perspective.

2.2.6 Computational Complexity

As we have mentioned, “Pattern Theory” might more appropriately be an un-named sub-set of computational complexity theory. The theory of computational complexity (e.g. [33, 54, 64]) has well developed measures of complexity. The measures used in Pattern Theory are a special case of these. There are also many computing theory results in what we call Pattern Phenomenology. However, complexity theory is oriented towards analysis rather than the synthesis of computational systems. Sections 4.4 and 4.5 develop the relationship between conventional measures of complexity and Pattern Theory.

2.2.7 Data Compression

The design of a data compression system depends upon recognizing and exploiting some pattern in the data. However, like algorithm design texts, data compression

texts (e.g. [27]) give you a list of specific procedures for some common patterns that were recognized by people. They do not tell you how to find new patterns in data.

2.2.8 Cryptography

Cryptography is concerned with patterns in sequences rather than functions. Although any mathematician will tell you that a sequence is a function, the problem is somewhat different. Pattern theory has so far been concerned with patterns in functions. Although the problem of breaking codes must involve pattern finding in the sense of Pattern Theory, we have not explored how Pattern Theory relates to cryptography.

2.2.9 Switching Theory

From a Pattern Theory perspective, the design of electronic circuits is essentially the same as algorithm design. Unlike algorithm design though, there are many theoretical synthesis tools. There seem to be three approaches to the design of discrete circuits. One approach (e.g. [21]), using ROM or PLA's, uses an essentially brute force table look-up. This approach offers no special insight into the pattern finding problem. A second approach is to design optimal two-level circuits [21]. This approach does not capture patterns in a sufficiently general sense because some highly patterned functions (e.g. the parity function) do not have efficient two-level realizations. The third approach is based on function decomposition. The idea of function decomposition has been around a long time (see [4]), but it has had a limited role in circuit design. Function decomposition is not even mentioned in many standard Switching Theory texts (e.g. [21, 26, 45]). When function decomposition is discussed (e.g. [60]), there seems to be general agreement that function decomposition is "prohibitively laborious." We believe that function decomposition gets at the crux of computational complexity. The practical difficulties of using function decomposition for circuit design does not detract from its central theoretical role. If nothing else, we hope that Pattern Theory will contribute to the realization that function decomposition is a (if not *the*) fundamental problem in computer science.

2.2.10 Summary

There are many disciplines that are relevant to Pattern Theory. As Pattern Theory matures there will be many potential areas of application. There are also many results from these related fields that are useful in Pattern Theory. We especially use some complexity ideas from computing theory and the function decomposition idea from switching theory.

Chapter 3

The Pattern Theory Paradigm

3.1 Why is Pattern Theory Needed?

This section will attempt to motivate the Pattern Theory work. This motivation is developed by picking a particular problem, discussing the importance of computing in solving this problem, discussing the role of algorithms in doing computing, and, finally, discussing the need for a theory to design algorithms.

3.1.1 Offensive Avionics as a Potential Application

The Pattern Theory work was performed in the Mission Avionics Division of the Avionics Directorate of Wright Laboratory (WL/AART). This organization has offensive avionics algorithms as a principal product. Therefore, we use this potential application of an algorithm design theory as an example to motivate the need for such a theory. The arguments used here could have been couched in terms of any one of the many diverse problems requiring algorithms (see Section 2.2). We chose offensive avionics algorithms because we are most familiar with this application and it helps explain why it is appropriate for Pattern Theory work to be done in this organization.

3.1.2 Importance of Computing Power in Offensive Avionics

Offensive avionics (or fire control) is responsible for locating, identifying and selecting targets, appropriately releasing weapons and doing this in the most survivable manner possible. In order to better understand what must be done to meet the responsibilities of fire control, we often think of fire control as a family of functions. These functions serve one of two purposes. Either they are part of the overall sensor system or they are part of the control system. The sensor system attempts to determine the "state-of-the-world," which includes targets, self, threats, cooperating friendlies and anything else that could be a factor. The control system manages all the resources of the aircraft. This includes deciding on the specific trajectory for the aircraft, managing

the sensors, as well as managing the weapons themselves.

All these functions have always been part of the fire control problem. At one time, the "weapon system" was just a person. This person formed their state-of-the-world picture from what they could see and hear. They moved into position on foot and instinctively planned and executed their "weapon delivery" (perhaps a punch or kick). Over time, people began to use artificial weapons, at first sticks and stones but eventually guns and bombs. We began to use artificial sensors such as telescopes and radars. We also developed artificial means of locomotion, beginning with horses and eventually leading to airplanes. We have added these increasingly sophisticated machines to a person always trying to improve the overall weapon system performance. Until recently, the extremely adaptive nature of people has allowed them to do their state-of-the-world assessment, their planning and control functions and to use these machines effectively. However, there has been an explosion in the complexity of the weapon systems. Now we not only have an artificial sensor, we have multiple sensors, each capable of measuring multiple attributes of many targets. Our artificial weapons now include many types, some with long range and many degrees of flexibility. Our means of getting about have become faster and more maneuverable.

At first we tried to deal with the increasing complexity by putting more people in the system. The crew size for bombers was six when we built the B-52. Then, as computers and software technology became available we began to deal with the complexity more and more through aids and automation. The crew of the B-1 was down to four and the B-2 has only two crew members.

What technology has allowed the crew size to decrease despite an increase in the complexity of the task? What technology may eventually allow the crew size to go to zero? The crew provides no useful work in the force times distance sense. Their sensory capabilities, in terms of being able to resolve and detect light, sound or acceleration could be easily replaced. People are in modern combat aircraft for one reason: their computing power. Therefore, it is fair to say that computing power is an extremely important technology for avionics systems.

3.1.3 Importance of Algorithms in Computing Power

In the preceding section we discussed the importance of computing power. Now we want to discuss how important algorithms are in overall computing power. We can think of computing power as being made up of three technologies. One technology is computing hardware. Fairly good measures of hardware capability exist in terms of Instructions per Second, Operations per Second, etc. There has been tremendous growth in computing hardware technology. In addition to hardware, effective computation requires software. We like to think of this software as being developed in two stages. First there must be some algorithm that describes the desired computation at an abstract level. Then this algorithm must be implemented in a specific computer language. We consider the first problem to be algorithm design and the second problem to be software engineering. These problems are not entirely separable, just as the hardware and software problems are not entirely separable; however, it is useful to

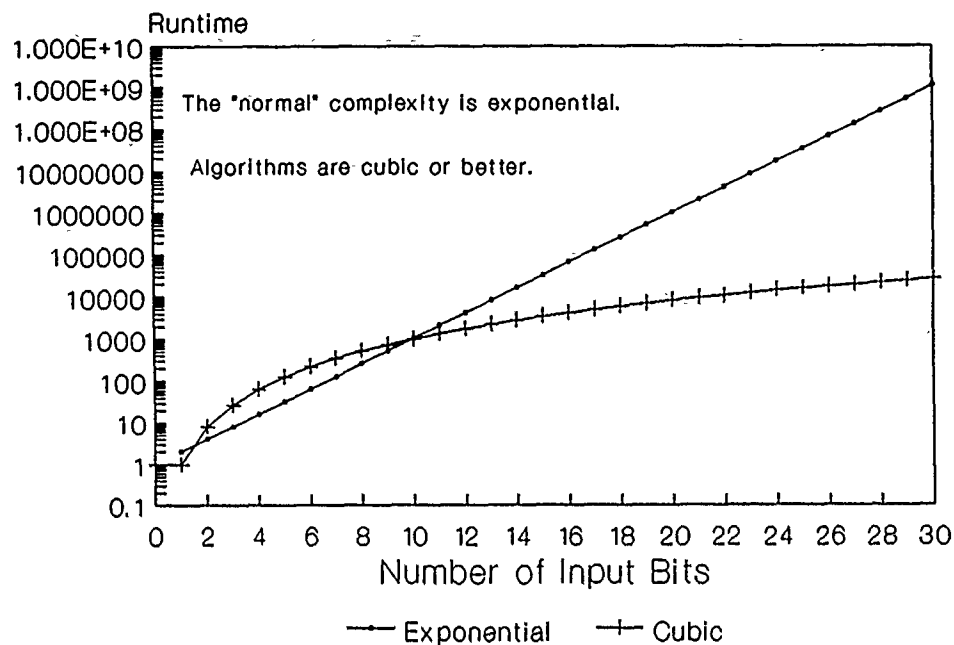


Figure 3.1: Time Complexity of Algorithms

break out algorithm design so we can concentrate on it without having to address specific implementations. There are some pretty good measures of algorithm goodness, but not for how well software engineering is doing nor for overall computing power. Although we cannot do so quantitatively, we still want to point out the special role of algorithm power.

The expected computational complexity (time complexity, program length, number of devices in a circuit, how ever you want to measure it) of an algorithm for an arbitrary problem goes up exponentially with respect to the size of the input (see Section A.2.4). Even poor algorithms are low order polynomial complexity. The difference between having even a poor algorithm and no algorithm becomes tremendous for problems of even modest size input (see Figure 3.1). Figure 3.2 shows typical input sizes for some problems of interest. Consider the difference between having and not having an algorithm and then consider the potential for hardware or software engineering to make up for this difference. Take an estimator as an example: not having an algorithm could only be compensated for by hardware or software engineering through an increased capability of $2^{(10^6)}$ times. This demonstrates how ridiculous it is to even think about computing most functions without a good algorithm.

Some specific examples further demonstrate the payoff in having a good algorithm. As mentioned earlier, estimation theory has an important role in fire control. In realizing most estimators it is necessary to invert a matrix. One test for whether or not the inverse of a matrix exists is to compute the matrix's determinant. To compute the determinant by standard recursion (as in the usual definition of the determinant) the run-time goes up as the factorial of the matrix size. However, with the Gauss-Jordan Elimination *algorithm* it is possible to compute the determinant with complexity $n \log n$ (see [7, 9]). As an example of what this means, if computing the determinant of a 20×20 matrix takes on the order of 50 milliseconds by the Gauss-

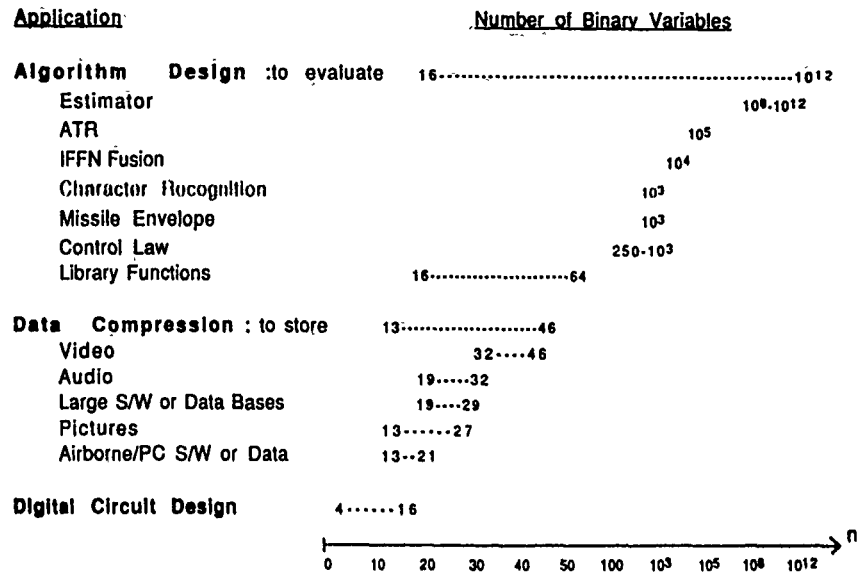


Figure 3.2: Typical Algorithm Input Sizes

Jordan Elimination method then it would take on the order of 10 million years by the standard recursion method. Sorting a list with Insertion sort has complexity n^2 (570 minutes to sort 100,000 elements) while Quick sort has complexity $n \log n$ (30 seconds to sort 100,000 elements). The relatively recent invention of algorithms like Quick Sort has brought about many of the word processing features that we use everyday. Even minor improvements in an algorithm can have dramatic effects. For example, the invention of the Fast Fourier Transform (FFT) algorithm only reduced the complexity from n^2 to $n \log n$ (see [46]). However, without the FFT algorithm, today's real time digital Synthetic Aperture Radar (SAR) capability could only be achieved with a hardware throughput improvement of about five orders of magnitude. Note that the FFT and Quick Sort algorithms were "invented." Without an engineering theory, things are invented. With an engineering theory, things are designed.

Therefore, good algorithms are very important in effective computing and in a real sense more important than hardware or software engineering.

3.1.4 Role of a "Design Theory"

We have gone from recognizing the need for computing power to the need for algorithms; now we want to recognize the need for an engineering theory to help design algorithms. But before we do that, we review the role of an engineering theory in design.

Although there are some particular well established engineering design theories (e.g. Modern Control Theory or Estimation Theory) there does not seem to be much literature on these kinds of theories in general. There is a body of literature on methods to improve the creativity of designers (e.g. [6, 17]). There is also some work on a theory about design (e.g. [28]). However, these do not treat "design theory"

in the desired sense. The most relevant literature about engineering design concerns "optimal design" (e.g. [47, 57]). In this literature the design process is one of defining a model, establishing the criteria for a good design and then optimizing the design with respect to that criteria. While most of the traditional optimal design theories have quantitative criteria and specific methods for optimization, they would be of value even without that. A good design theory tells you what is important about a class of problems, tells you about some absolute limits on performance, allows you to predict performance, and gives you some specific steps towards solving a class of problems. For example, estimation theory (e.g. [36]) tells you that it is important to model the dynamic behavior of the system (i.e. $\dot{\vec{x}} = A\vec{x} + B\vec{u}$) and the measurement process (i.e. $\vec{z} = H\vec{x} + G\vec{w}$) as well as the specific form of an optimal estimator based on these models. Estimation theory also allows you to determine any observability limitations. Ideally, a design theory would have a formal structure. As Melsa and Cohn [39] say in regards to decision and estimation theory:

"Although we treat such problems intuitively all the time, it is important that we cast them into a more definite mathematical model in order to develop a rigorous structure for stating them, solving them, and evaluating their solution."

By a mathematical model we would not necessarily mean a numerical model, only that the model have a formal logical structure.

A good design theory is not the solution to any particular problem; rather, it is a tool useful in solving a whole class of problems.

3.1.5 The Need for a Design Theory for Algorithms

Historically, Electrical Engineering design theories (especially estimation and control theory) have been used to develop fire control algorithms. However, the modern fire control problem requires a large variety of algorithms. Many of these problems are either not naturally representable as estimation or control problems or the solutions provided by these traditional theories are computationally intractable. For example, the determination of an aircraft trajectory for attacking multiple ground targets in a single pass can be set up as an optimal controls problem. However, because closed form optimal solutions cannot be found, this leads to a computationally impractical design. Further, the problem of selecting a trajectory for the attack of multiple airborne targets cannot even be set up as a reasonable controls problem. The point we are trying to make is that there is a need for a more general theory of algorithm design. Our recognition of this need arose in considering fire control problems but the need is pervasive in the application of computing power.

With the extensive literature on algorithms it seems surprising that there is not a general theory of algorithm design. However, most of this literature is concerned with the analysis of algorithms rather than their design. Even the literature on algorithm *design* typically does not discuss how to create an algorithm; rather they tell you how to apply known algorithms in various situations. When algorithm creation

is discussed, it is in terms of "discover" or "invent" rather than design (e.g. "The 'discovery' by Cooley and Tukey in 1965 of a fast algorithm ..." [7] or "The *creation* of an algorithm ..., is an inventive process ..." [38]).

Once connected with the problem of "discovery", we begin to wonder if a design theory for algorithms is even possible. Reference [31] argues that it is not only possible to have a theory of the discovery process but that it is possible to automate the process. While we think they are correct, this report is concerned with simply trying to *understand* algorithm design in a formal theoretical sense. We feel that a thorough understanding of the problem is the first step to a useful design theory and that design assisted by an engineering theory would logically precede automation of algorithm design.

3.1.6 Summary

This section attempts to show the practical relevance of Pattern Theory. We began by discussing the importance of computing in offensive avionics; although the importance of computing could have been derived from many sources. We then point out the special dependence that computing power has on algorithm design. Improved hardware or software engineering are fine tuning compared to new algorithms which create entirely new capabilities. After clarifying what we mean by a "design theory," we explain that a design theory for algorithms would be very beneficial and that such a theory does not currently exist. The bottom line is that there is a strong, un-met, need for a theory of algorithm design.

3.2 The Pattern Theory Approach

Pattern Theory is an attempt at an engineering design theory for algorithms. This section will present the algorithm design problem in a way consistent with an engineering theory. We begin by first expanding on our concept of a design theory.

3.2.1 The "Given and Find" Characterization of a Design Theory

We will develop our concept of a design theory in terms of "givens" and "finds." Given and Find are intermediate stages in going from the real problem to the real solution. The design theory provides methods for relating the given problem statement to what we want to find. However, there always remains the task of couching the real design problem into a simplified problem of specific givens and finds such that the design theory can be applied.

Many engineers first encounter a design theory in Statics. Therefore, we use a problem from statics as our example, from [40]. The real problem is to design a roof that will support whatever snow, wind, etc. that will stress it. The first step in going from the "real problem" to the "given" for the design problem is to select some form

of truss. For example, a Howe truss could be selected. This selection might be based on the designer's recognition that it is appropriate for this class of problem, but is outside the design theory. A second step in going from the "real problem" to the "given" for the design problem is to make some assumptions about the loads that will be applied to the truss. These assumptions take the form of a certain magnitude force applied at certain points on the truss. These assumptions might be based on the designer's knowledge of local weather, etc.; but again, this is outside the design theory. We have gone from the "real problem" to a set of "givens." This part of the design is not based on any "design theory," rather it depends upon the human element in design.

The "real solution" to this problem might consist of a complete specification of materials in the truss, the size and shape of the members of the truss, how the members are joined, etc. The designer recognizes that if the forces in the members can be found, then it would be easier to complete the real problem. For example, a catalog could be used to select truss members once the maximum load on a given member was known. Therefore, we say the "find" is the force in each member. Again, going from the "find" to the "real solution" will not be aided by the design theory. However, now that we have specific "givens" and "finds," we can apply the "design theory" of Statics to connect these two. In particular, given the loads on a particular truss we can solve for the forces in each member of the truss.

In summary, a design theory operates within the simplified environment of specific "givens" and "finds." The messy problems of determining the "givens" from the real problem and the real solution from the "finds" are outside the theory. Pattern Theory is an attempt at a design theory in this sense for algorithms.

3.2.2 Definition, Analysis and Specialization

It is important that a design theory begin with a well-defined problem. Charles Kettering is reported to have said:

"A problem well stated is a problem half solved."

Our approach to stating the problem is to first define a very general and abstract problem (Section 3.3). A problem is well-defined when we can say precisely what is given, what is to be found and the criteria by which the solutions are to be judged. The problem will then be analyzed to determine how it might be partitioned into simpler problems. Finally, we specialize to one of the simpler problems (Section 3.4.9). We deliberately and explicitly set aside some aspects of the problem. There are two purposes to this approach. First, it allows us to arrive at a well-defined and potentially solvable problem. Second, it allows us to understand how our problem is a special case of more general problems.

3.3 The General Problem of Computational System Design

Here we develop the most general Pattern Theory problem. This problem is a central part of many disciplines (c.f. Chapter 2). First we must deal with several rather general, almost philosophical, issues. We will explain why we are especially interested in recognizing patterns in functions, the meaning of a representation of a function, and figures-of-merit for competing designs.

3.3.1 Computation and Functions

We can imagine trying to recognize patterns in all kinds of mathematical objects. The examples of Section 2.1 were typically sequences. However, we believe that functions have a unique importance when considering pattern finding in connection with computation.

First of all, functions are a fundamental mathematical concept. A function f is a set of ordered pairs from $X \times Y$ such that for all (x_1, y_1) and (x_2, y_2) in f , if $x_1 = x_2$ then $y_1 = y_2$. This definition of a function only requires some set theory, order, and logic as background.

The only trick to being a function is that there be only one output for any given input. For example, in an Automatic Target Recognition setting our assumption is that there is exactly one desired output (e.g. target type) for each input (e.g. an image). This assumption does not preclude the output from having probabilities; in this case our assumption only requires that there be exactly one desired output probability distribution (e.g. $p(\text{tank}) = 0.1$, $p(\text{truck}) = 0.6$, $p(\text{tree}) = 0.2$, ...) for each input. Our assumption does preclude those cases where there are a significant number of inputs for which multiple possible outputs would be acceptable. For example, if we consider outputs of either 0.99 or 1.0 to be acceptable then our assumption does not hold. While this may seem to be the more common situation, it is possible to define a codomain for almost any real problem such that the assumption does hold. For example, we could define "0.99 or 1.0" as a single output value.

Functions are also abstractions of most of the traditional models of computation. Language acceptance is a common model for computation in the theory of computing. Language acceptance is a special case of a function; that is, a language acceptor is a function from a set of strings into the binary set {accept, reject}. Problem solving is a common model of computation in computing theory and some artificial intelligence contexts. Problem solving is a function from a set of problem definitions into the set of possible solutions. Decision making is also a function from the factors in the decision into the set of possible decisions. Functions are a "show me" approach to modeling knowledge. What a computer (or person) knows is exactly the set of questions that it can answer. We would say that knowledge is well represented by a function from a set of questions into a set of answers. Reference [49] discusses this relationship between mathematical functions and knowledge at length. Many models of machine

learning, e.g.[13, p.326] or [42, p.6], can also be interpreted as special cases of function realization.

Non-function computation problems exist, such as the generation of one-way communications (radio/TV) or clocks, but virtually all conventional computing is well modeled by functions.

In summary, a function is an extremely general and well-defined model for computation. When we talk about computation we are talking about realizing a *function*.

3.3.2 Representation

The notion of representation is very important in Pattern Theory (see [49, pp.29-50]). The design problem begins with some sort of a representation of a function and then we want to find an efficient algorithm that will also be a representation of that same function. Therefore, the design problem is one of translating representations.

The representation of a function is meaningful only if there is some agreed to "representation system." The representation system is kind of like the syntax and semantics of a language. It is the background knowledge that one must have to make sense of a representation.

We do not have a formal definition of a "representation system." Think of representation in the sense of communication. Whenever we represent a function, we must assume that the reader has some knowledge that allows them to make sense of the representation. This "knowledge" is what we are trying to specify with "representation system." An important and unsolved problem of Pattern Theory (and computing in general) is that of dealing with this idea of a representation system. Sections 3.4.3 and 3.4.4 explain how we get around this problem for the PT 1 project.

The representation system used for defining the function to be computed is called the "input representation system." Input comes from this being the input to the design problem. PT 1 focused on tabular input representation systems. The representation system used for the solution is called the "output representation system." Again, output comes from this being the output of the design problem. PT 1 used directed graphs with functions at each node for the output representation system.

In addition to the concept of a representation system, there are many forms of representation within each system. Several classes of representation are identified as examples of this idea.

First, there is the simple *table* definition of a function (e.g. Table 4.1). A table seems to require the minimum possible representation system.

Secondly, there is the class of *algorithmic* representations of a function. These representations give an algorithm for computing $f(x)$ when given x . The representation $f(x) = x^2 + 2x - 1$ is algorithmic. The representation system for this example must include knowledge of arithmetic. A common situation in fire control algorithm design is to have an algorithmic definition of a problem (often called a "truth-model") that is too slow for airborne use. The design problem is to find a *better* algorithmic representation.

A third class of representation might be called the *algorithmic inverse* class. Representations from this class provide algorithms that, when given y , produce x such that $y = f(x)$. An example of an algorithmic inverse representation of f is $x = y^2$, where $y = f(x)$. Therefore, when given y we can generate x using the representation; however, the representation does not explicitly tell us how to generate y when given x . The "vision" problem is a more practical example of an algorithmic inverse representation. For the vision problem, the function that we want to realize (i.e. a mapping from a two-dimensional image into a three-dimensional model of a scene) is easily represented in inverse form. That is, we can use geometric projection, which is algorithmic, to determine what two-dimensional image would result from a given three-dimensional scene.

Our fourth class of representation is the *algorithmic NP* class. The "NP" comes from the non-deterministic polynomial set of functions as studied in time complexity which have this form of representation. For an algorithmic NP representation, we must be given both x and y and then the representation is an algorithm that will determine if $y = f(x)$. An example of this class of representation is when the function has some equation as its input and solutions to the equation as its outputs. When given the equation and a candidate solution, it is easy to tell if the solution fits.

A fifth class of representation is called the *function predicate* class. In this class, the function is represented by some algorithmic predicate on the whole function. An example of this class is a differential equation.

A final class might be any mix of the above classes. For example, a function can be represented by a differential equation (function predicate class) and boundary conditions (table class).

3.3.3 Figures-of-Merit

There is one other idea that needs to be developed before we can state the general problem. This idea has to do with what constitutes a "good" design. A well designed computational system should have a number of properties. We divide these properties into two general categories.

One category has to do with the accuracy of the computational system. That is, how often does it produce errors or no output at all. Errors could be defined as the difference between the desired function and the function actually computed. There are many options for defining the difference between functions. For example, if X has finite cardinality and Y is the set of real numbers then the difference (d) between functions $f : X \rightarrow Y$ and $g : X \rightarrow Y$ might be $d = \sum_{x \in X} |f(x) - g(x)|$. For many computational problems, we want no errors. For other problems, avoiding all errors is either simply not possible or not worth the increased cost.

The second category of properties concerns monetary costs. There are costs associated with arriving at the design, physically realizing the design and using the design. In arriving at the design there are the costs of gathering samples of a function or of performing experiments to narrow the possible set of functions. We associate these costs with the definition problem (see [50]). The cost of realizing a design includes the

cost of purchasing and assembling equipment. This is the cost of concern in circuit design (e.g. [21]). The cost of using the design is most often thought of in terms of the run-time or memory use on a sequential computer (e.g. [7], but is also reflected in circuit design as "depth." While there is considerable latitude for trading-off equipment cost versus run-time, we think this trade-off is between different ways of exploiting the singular pattern-ness of a function rather than between different kinds of patterns. Therefore, we want our measure of pattern-ness to be high whenever it is possible to realize a function with low equipment cost and reasonable run-time or with small run-time and reasonable equipment cost.

3.3.4 Problem Statement

We now state the general computation system design problem of PT. The statement of the problem is not sufficiently precise to be useful in the design theory sense. The purpose in stating this general problem (a problem that includes virtually everything anybody does with computational systems) is that it will allow us to show how the PT 1 problem (Section 3.4) is a special case of the general problem.

We state the problem in terms of what is given and what is to be found. For the general problem, we are given an input representation system, a set of functions represented in the input representation system, a set of output representation systems and figures-of-merit.

The "input representation system" is the language in which the function(s) to be computed is given. Sometimes, if there is a precise definition of the function, the input representation system might be little more than arithmetic. For example, the function might be given as: "compute y when given x where $y = x^2 + 2x + 3$." However, when the function is given in vague terms, the representation system might include a natural language as well as many value judgements. For example, a function might be given as: "compute y when given x , where x is time and y is the intensity and color of the video signal of a new hit TV show." Although it may always be difficult, and sometimes impossible, to specify the input representation system, we think that such a characterization is a necessary step towards a theoretical engineering understanding of the problem.

In addition to the input representation system, there must be representations (expressed in the input representation system) that define the specific functions that we want to compute. In our general statement of the computational system design problem we allow for there to be a set of functions to be computed rather than just a single function. We can imagine that when computing several functions, the computation of one function might be used in computing a second function. Therefore, the design problem is somewhat different when computing more than one function. It turns out that it is not as different as we once thought (see Section 6.2.2).

The design of a "general purpose" computer requires that the most general problem not only allow for a set of functions, but that there be some super set of functions and that we do not know which exact subset is to be computed. The idea here is that there is some set of functions that you might potentially want to compute, but

you do not know exactly which ones. Therefore, the design problem is to come up with the computer that would do well on average for any subset of functions that might be specified later. This is the problem faced by people who design general purpose computers. As with the representation systems, it is not easy to specify the set of given functions but this specification is necessary for a theoretical engineering treatment of the problem. Designing algorithms or electronic circuits is a special case where the functions to be computed are known ahead of time.

The output representation system is the representation system that will be used to express the design. For circuit design (including the design of general purpose computers) the output representation system typically consists of some set of circuit elements. In algorithm design the output representation system might be a particular computer language. We said that the "givens" might include a *set* of output representation systems. Why a set? In the most general design problem we include the problem of selecting the output representation system. By specifying an output representation system, we are limiting the scope of possible solutions. Limiting the scope of solutions is not desirable in itself but is necessary for an engineering theory of the problem. This scope limiting part of the problem specification is characteristic of other engineering design theories. For example, classical control theory limits consideration to control laws based upon linear combinations of the system states and the desired states.

The figures-of-merit reflect error, the cost of the output representation systems and the cost of individual representations. Differences between the given function and the realized function is what we are calling "error." We sometimes do not insist that the error be zero. Instead we want it to be close but not to the point of compromising the other considerations (especially cost). Therefore, the "givens" must reflect our relative tolerance for errors and cost. The cost of the output representation system is essentially the cost of the computer hardware. The cost of the individual representation is sometimes the monetary cost of the hardware (as in circuit design) and sometimes the cost of execution (for example the run-time of an algorithm).

We stated the problem in terms of what is given and what is to be found. For the general problem, we are given an input representation system, the representations of a set of functions, a set of output representation systems and figures-of-merit. The problem then is to find an output representation system (from the set given) and the representations of a subset of the given functions such that the figures-of-merit are optimized.

3.4 The Pattern Theory 1 Problem as a Special Problem in Computational System Design

Recall that the objective is to isolate that part of traditional algorithm design that depends on this special character of patterns that we have discussed. This problem will be analyzed and the results extended back towards more practical problems.

There are two basic mechanisms for doing this isolation. First we can partition the general problem into sub-problems, allowing us to set aside some very difficult practical problems that are not directly involved in the pattern issues. The retained portion of the partition will have the pattern issues more accessible. The second mechanism is to simplify the general problem, always retaining a non-trivial pattern finding problem.

3.4.1 Special Rather Than General Purpose Computers

The Pattern Theory (PT) 1 problem is concerned only with realizing a set of functions that are known ahead of time. As discussed above, when designing general purpose computers, we do not know what exact functions we will eventually be computing. The design of "special purpose" computers includes circuit design as well as specific uses of general purpose computers. Therefore, our sense of "special purpose" computer design includes algorithm design.

3.4.2 Single Function Realization

For the general problem we allowed for there to be several functions to be realized. The PT 1 is concerned with realizing only a single function. Realizing a single function seemed to be a simpler problem that still requires pattern finding in a non-trivial sense. It turns out that it is not possible to get completely away from realizing multiple functions because when you decompose a single function you are creating multiple "sub-functions" that must be realized. The Lupanov representation (see [54, pp.116-118]) re-uses computations of sub-functions in realizing individual functions. However, for the relatively small number of variables considered in the PT 1 study, this re-use technique is not effective. Therefore, it is meaningful to consider single function realization as a further specialization to the general computing problem.

3.4.3 Input Representation System

For the general computing problem, we did not specify a particular input representation system. In fact, we did not even give a formal definition of a representation system. For the PT 1 problem we chose to limit consideration to input functions represented as tables. The representation system for tables is trivial; that is, the knowledge required to use a table is trivial. This specialization allows PT 1 to avoid having to deal with the messy problem of input representation systems. However, the pattern finding problem when given a function in the form of a table is not trivial. In fact, this is kind of a worst case for pattern finding. When a function is specified in some non-trivial representation system there may be some clues as to the patterns in the function. However, when the function is given as a table, there are no clues. As desired, this specialization results in a cleaner theoretical problem while retaining the essential pattern finding problem.

3.4.4 Output Representation System

The output representation system for PT 1 will be kept at a fairly abstract level. Therefore, selecting an output representation system does not require that we select a specific programming language or a specific set of circuit elements. The output representation system for PT 1 is a directed graph with a function associated with each node. The details are defined in Chapter 4. On the surface it may appear that PT 1 specializes to combinational machines with a loss of applicability to sequential machines and, in fact, we do represent our decompositions combinatorially. However, because of the connection between time (sequential) complexity and size (combinational) complexity, the patterns found in the decomposition process are not essentially different than sequential patterns (see Section 4.4). Therefore, although PT 1 does specialize to combinational output representation systems, it does so without loss of generality.

3.4.5 Functions

PT 1 makes a number of specializations to the kind of functions considered. First of all, PT 1 is only concerned with finite functions. Although it has been useful to use infinite functions (to the exclusion of finite functions, i.e. all finite functions are computable and have complexity $O(1)$) in most traditional computing theory paradigms, there is no greater generality in infinite functions. It is simply a matter of convenience. We feel that any real problem can be modeled finitely, whether or not the solution is eventually implemented in an analog or discrete system. Therefore, although unusual, our specialization to *finite* functions is without loss of generality.

We are especially interested in mappings on domains whose elements have parts. That is, the inputs are made up of multiple parts. There are two common models for these multiple part inputs, the string and the vector. Vectors are thought of as elements of a product of sets (as in $X_1 \times X_2 \times X_3 \times \cdots \times X_n$). All the vectors in a set typically have the same length, that is dimension, not metric length. Infinite-dimensional vectors are commonly used in Real Analysis. Strings are thought of as a sequence of drawings from a single set. Strings are typically not all the same length. Strings may also have infinite length. Either is sufficiently general to model the other. All strings of length n or less from an alphabet Σ can be modeled as vector elements of $(\Sigma \cup \{blank\})^n$, where vectors with a blank left of a non-blank are not included. Similarly, vectors from $X_1 \times X_2 \times X_3 \times \cdots \times X_n$ can be modeled as those strings of length n from the alphabet $\Sigma = \cup_{i=1}^n X_i$, with the i^{th} component from X_i . Vectors are especially common models in Electrical Engineering applications such as Circuit Design, Estimation Theory, Control Theory, and Digital Signal Processing. String based models are especially common in Computer Science, e.g. compilation problems, and are used in Computability Theory. Vectors have slightly more transparent combinatorics (see Appendix A). There are approximately twice as many strings as vectors for a given maximum length. While this difference might be important in some particular instance, the trends are the same for functions on a

set of vectors as for functions on a set of strings. The ideas in this report could be developed exclusively in terms of strings or vectors; without any significant difference in the fundamental results. We use vectors in the quantitative discussions because of their simplified combinatorics; however, we also use the string nomenclature in order to highlight relationships to the traditional theory of computing.

In addition to limiting consideration to finite functions, PT 1 limits consideration to binary functions. Binary functions are functions of the form $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Functions on any other finite domain can be modeled as binary functions. Also, functions with other codomains can be modeled with multiple binary functions. Therefore, there is little loss of generality in limiting consideration to binary functions.

3.4.6 Definition versus Realization

In reference [51] we made a distinction between the problem of choosing what function to compute (the definition problem) and the problem of figuring out how to compute the chosen function (the realization problem). Even in our general statement of the computing problem we had already limited consideration to the realization problem. The Pattern Theory idea is to associate patterns and simplicity. Patterns are those functions with economic realizations; these may be called "realization patterns." There may also be, in this pragmatic sense, "definition patterns." That is, a function has a definition pattern if it is easy to define, for example, amenable to interpolation. It is not clear whether or not these two concepts of pattern can be unified; but for PT 1, pattern is addressed only in the realization pattern sense.

3.4.7 Figure-of-Merit for the PT 1 Problem

The figure-of-merits for the general problem included a consideration for error in the realized function. This factor is important in real computational system design (see [61]), but perhaps not as important as we once thought (see Section 6.6). PT 1 limits consideration to exact realizations (as in [48, p.33]). This specialization is made at the expense of generality to allow us to focus on the pattern issues. Although finding the pattern in a function that allows its exact computation is a special case of finding the patterns that allow functions to be approximately computed, it is by no means trivial. We think the essential character of the pattern finding problem is preserved in the PT 1 problem and made more readily accessible by setting aside the error issues. Section 6.6 further discusses the cost-error trade-offs.

The general problem also included several measures of cost (hardware costs, runtime, etc.). A central thesis of the Pattern Theory paradigm is that all these costs are well represented (with respect to our pattern finding problem) in one abstract measure, Decomposed Function Cardinality. Chapter 4 explains and supports this thesis. Therefore, the PT 1 problem specializes to this single measure with very little loss of generality.

3.4.8 Kinds of Patterns

We discussed previously that functions have a property called "realization pattern-ness." That is, some functions have simple realizations while others do not. Of course, whether or not a function has a simple realization is relative to the representation system. Reference [48] develops the idea that there are two fundamental mechanisms that allow simple realizations. That is, there are two kinds of realization patterns. One kind is based on the relationship between a function and physical processes (call this kind "physical patterns"). The second kind is based solely on the decomposability of the function (call this kind "decomposition patterns"). PT 1 is concerned only with this second kind of realization patterns.

There is some definite loss of generality here. Some functions are naturally realized by some devices and exploiting this is essential in some real problems, for example the computation performed by an optical lens. However, applications of general purpose computers seem to rely on decomposition patterns. Functions with physical realizations may also have decomposition pattern-ness (e.g. addition can be realized physically by many means and addition also has high decomposition pattern-ness). There seems to be a problem here for the physicists to worry about. Why does the natural world have such a high degree of decomposition pattern-ness?

3.4.9 Problem Statement

Recall that the objective is to isolate that part of traditional algorithm design that depends on this special character of patterns that we have discussed.

Through the specializations discussed above, we arrive at the PT 1 problem. That is, given a single finite binary function completely defined as a table, find an exact combinational realization that minimizes the Decomposed Function Cardinality measure.

We have gone from a very general, but somewhat vague statement of computational system design to a less general but definite statement that retains the essential pattern finding problem.

3.5 Summary

This chapter explained the need for an algorithm design theory and defined the Pattern Theory paradigm as a potential approach.

The need for an algorithm design theory was shown by starting with the obvious need for offensive avionics, showing how important computing power is in offensive avionics, showing how important algorithm design is in computing power and then finally showing how important an engineering theory is to design. Although this development could have been based on many different kinds of problems, the importance of an algorithm design theory to offensive avionics is sufficient to justify our research.

The Pattern Theory paradigm was defined by first discussing the form of a problem definition for an engineering theory. We then stated the most general problem that

might be characterized as "computational system design." Finally, we made explicit specializations to this general problem to arrive at the PT 1 problem, i.e. the problem of designing exact combinational realizations of binary functions, given as a table, such that the Decomposed Function Cardinality (DFC) is minimized. The PT 1 problem is much simpler than the general problem; yet it retains the essential pattern finding problem that is our focus.

Chapter 4

Decomposed Function Cardinality as a Measure of Pattern-ness

4.1 Introduction

The central thesis of Pattern Theory is that function decomposition is a way to get at the essential idea of computational complexity. The connection between decomposition and computation has come up before. The “divide and conquer” principle (e.g. [7, p.3]) is essentially a suggestion that decomposition is a good idea for algorithm design. The “chunking” model of learning (see [43]) is a form of function decomposition. The Abductive Reasoning paradigm represents functions by compositions of simpler functions. Function decomposition is also a generalization of representations that use arithmetic or logical operators. When we were first exposed to pattern recognition and machine learning, we were impressed with the prominent role of arithmetic operators, e.g. “...the adaptive linear combiner, the critical component of virtually all practical adaptive systems.”¹ We asked ourselves, “What makes arithmetic so special?” Why should it have any special powers? We now believe the answer lies in the fact that arithmetic operators are members of a common class of decompositions. However, there are other classes and that is why it is important to understand decomposition as the underlying principle with linear combiners as a special case.

We propose Decomposed Function Cardinality (*DFC*) as a quantification of a function’s pattern-ness. Section 4.2 contains a formal definition of *DFC*. Informally, we base our measure on the cardinality of a function. A function is a *set* of ordered pairs and, as with any set, a function has some cardinality. That is, for finite functions, a function has some number of elements. Function *h* of Table 4.1 has cardinality 8 while functions *f* and *g* of Table 4.2 have cardinality 16.

Now we need to distinguish between functions of the same cardinality that have different pattern-ness. First recognize that some functions can be represented as a composition of smaller functions. For example, *f* in Table 4.2 can be written

¹From the UCLA Adaptive Neural Network and Adaptive Filters Course announcement, Bernard Widrow and Mark A. Gluck Instructors, 1991.

x_1	x_2	x_3	$h(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 4.1: Function Cardinality of h is 8

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$	$g(x_1, x_2, x_3, x_4)$
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	1
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	0	0

Table 4.2: The Function Cardinality of f and g is 16

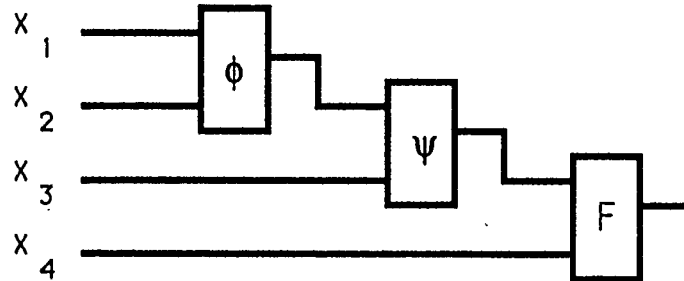


Figure 4.1: f as a Composition of Smaller Functions

z_1	z_2	$F(z_1, z_2)$	$\phi(z_1, z_2)$	$\psi(z_1, z_2)$
0	0	1	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1

Table 4.3: Functions that Compose f

$f(x_1, x_2, x_3, x_4) = F(\phi(\psi(x_1, x_2), x_3), x_4)$. This representation of f can be diagrammed as in Figure 4.1 with F, ϕ and ψ defined in Table 4.3. Notice that the cardinality of F, ϕ and ψ is 4 each. The sum of their cardinalities is 12. Therefore, f , a function of cardinality 16 can be represented by a composition of functions whose combined cardinality is only 12. We say that f has a Decomposed Function Cardinality of 12. Most functions, such as g in Table 4.2, cannot be composed from smaller functions in this way. Some functions have more than one decomposition; a decomposition is a representation of a function as a composition of smaller functions. When a function has more than one decomposition, DFC is defined to be the minimum combined component cardinality of all the decompositions of that function. The familiar decomposition of addition would look like Figure 4.2 with components as illustrated in Tables 4.1 and 4.1. The cardinality of a_1 and c_1 is 4 each, the cardinality of a_2, a_3, c_2 and c_3 is 8 each; therefore, the DFC of adding two numbers of three bits each is $4 + 4 + 8 + 8 + 8 + 8 = 40$.

The palindrome acceptor on six variables is a function with cardinality 64 (see Table 4.1). Figure 4.3 is a decomposition of a palindrome acceptor with Tables 4.1 and

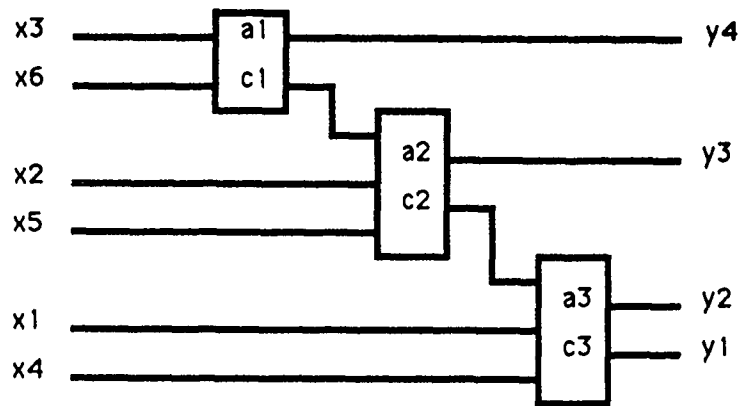


Figure 4.2: Decomposition of Addition

$x_1x_2x_3$	$x_4x_5x_6$	y_1	y_2	y_3	y_4
000	000	0	0	0	0
000	001	0	0	0	1
	\vdots				
010	110	1	0	0	0
	\vdots				
111	110	1	1	0	1
111	111	1	1	1	0

Table 4.4: Addition on Six Variables (Four Output Functions)

x_3x_6	a_1	c_1
00	0	0
01	1	0
10	1	0
11	0	1

Table 4.5: Addition Components a_1 and c_1 (XOR and AND)

$c_1 x_2 x_5$	a_2	c_2
000	0	0
001	1	0
010	1	0
011	0	1
100	1	0
101	0	1
110	0	1
111	1	1

Table 4.6: Addition Components a_2 and c_2

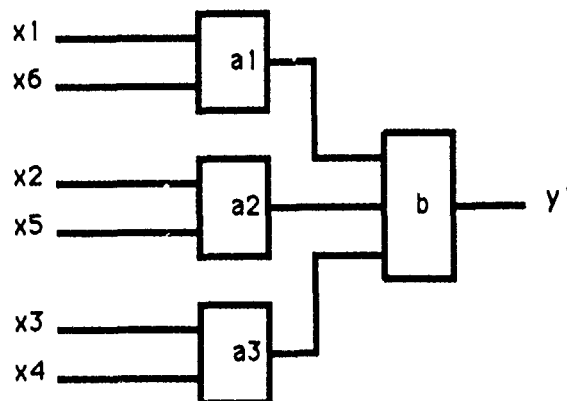


Figure 4.3: Decomposition of a Palindrome Acceptor

4.1 showing typical components. The *DFC* of the palindrome acceptor on six variables is $4 + 4 + 4 + 8 = 20$.

Figure 4.4 is a decomposition of the prime number acceptor on 9 variables with Table 4.1 showing typical components. The *DFC* of this particular decomposition for the prime number acceptor for inputs between 0 and 511 is 344.

Figure 4.5 is a decomposition of the function which declares whether a pixel should be black or white given the coordinates of that pixel for the 16×16 pixel image in Figure 4.6. Tables 4.1 through 4.1 define the components of this decomposition. The *DFC* of the "R" function is 36. Note that variables x_4 and x_8 are not needed.

All these functions have *DFC*'s less than the cardinality of the undecomposed function. We would consider each of these functions to be patterned. The palindrome acceptor ($[f] = 2^6 = 64$ versus *DFC* = 20) and the "R" function ($[f] = 2^8 = 256$

$x_1x_2x_3x_4x_5x_6$	y_1
000000	1
000001	0
000010	0
\vdots	\vdots
001011	0
001100	1
001101	0
\vdots	\vdots
011101	0
011110	1
011111	0
\vdots	\vdots
111101	0
111110	0
111111	1

Table 4.7: Palindrome Acceptor on Six Variables

x_1x_6	a_1
00	1
01	0
10	0
11	1

Table 4.8: Palindrome Acceptor Component a_1 (NOT XOR)

$a_1a_2a_3$	b
000	0
001	0
010	0
011	0
100	0
101	0
110	0
111	1

Table 4.9: Palindrome Acceptor Component b (AND)

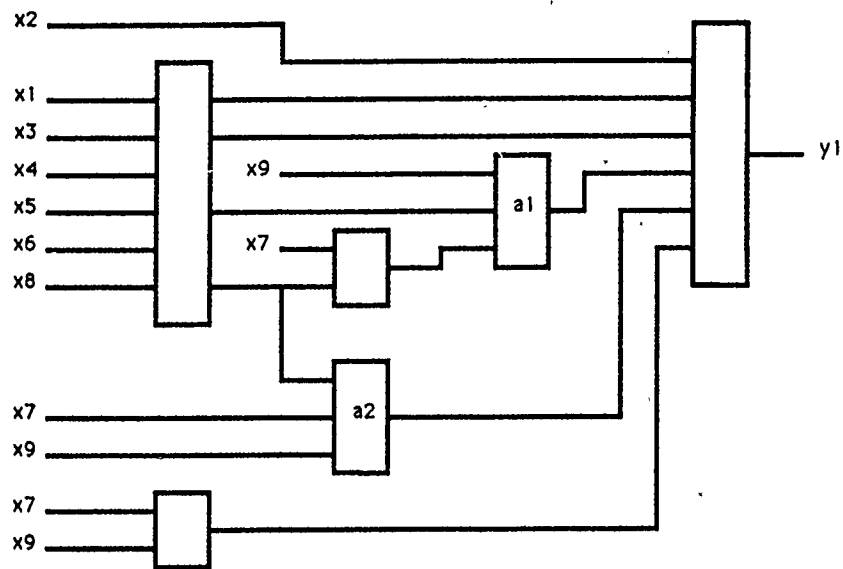


Figure 4.4: Decomposition of a Prime Number Acceptor

input	a_1	a_2
000	0	0
001	1	0
010	1	1
011	1	0
100	0	0
101	0	0
110	1	0
111	1	1

Table 4.10: Prime Number Acceptor Components a_1 and a_2

$x_5x_6x_7$	c	d
000	0	0
001	1	0
010	1	0
011	0	0
100	1	0
101	1	0
110	1	0
111	0	1

Table 4.11: Letter R Components c and d

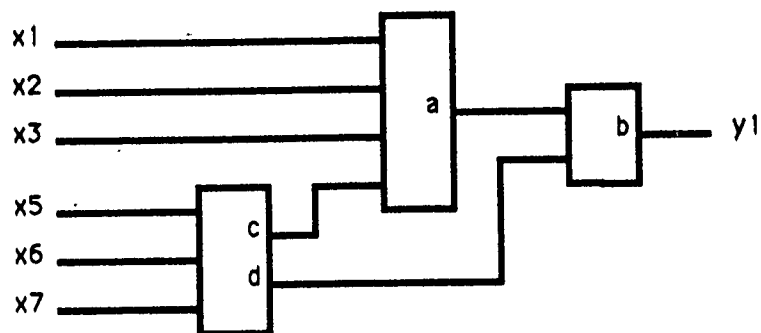


Figure 4.5: Decomposition of an Image

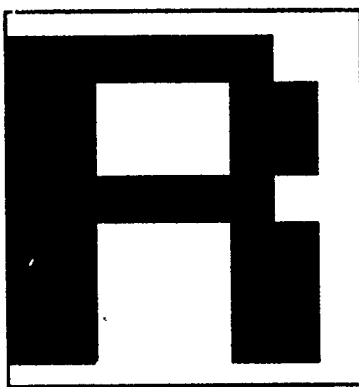


Figure 4.6: An Image of "R"

$x_1x_2x_3c$	a
0000	0
0001	0
0010	0
0011	0
0100	0
0101	1
0110	0
0111	1
1000	0
1001	1
1010	0
1011	0
1100	1
1101	0
1110	1
1111	1

Table 4.12: Letter R Component a

ad	b
00	1
01	0
10	0
11	0

Table 4.13: Letter R Component b

versus $DFC = 36$) are very patterned, while the prime number acceptor ($[f] = 2^9 = 512$ versus $DFC = 344$) is only slightly patterned.

The components of a decomposition can be any kind of function, not just the usual logical functions (AND, OR, XOR, etc.). Note also that some of these decompositions are very familiar (e.g. addition or the palindrome acceptor) while others were not previously known (e.g. the primality test or the "R").

We believe that DFC is a very robust measure of pattern-ness. As in the previous examples, where completely different kinds of patterns are involved, a small DFC (compared to the cardinality of the undecomposed function) is truly indicative of pattern-ness.

The development of DFC began at the Air Force Institute of Technology (AFIT) where Matthew Kabrisky stressed the fundamental importance of features in recognizer design. DFC resulted from an attempt to generalize the geometric pattern recognition concept of "features." We realized that features are just intermediate stages in the data flow that promote a computationally efficient realization of a function. Function decomposition also produces these intermediate stages of data for exactly the same purpose. DFC allows for an explanation of pattern-ness that covers the common pattern recognition paradigms, i.e. geometric, syntactic and artificial intelligence approaches [4⁹]. It seems that the one common factor in all these approaches was this decomposition idea. The idea gained further credence when we realized that the "Divide and Conquer" approach to algorithm design is essentially a function decomposition approach. We were eventually directed² to some theoretical developments of function decomposition in the Switching Theory literature. Here again is this idea in yet another context.

In addition to these informal arguments for the robustness of DFC as a measure of pattern-ness there are several more objective supporting arguments. Chapter 6 empirically relates DFC to factors in data compression and to complexity as rated by people. The remainder of this chapter will report on the relationship between DFC and three of the most common measures of computational complexity. First we consider program length. By treating DFC as a component of program length we can apply some useful results from Information Theory (see [12, 23]). We then relate DFC to time complexity. Under reasonable assumptions one can prove that a small time complexity implies a small DFC (see [54]). Finally, we point out that DFC is a special case of circuit size complexity. By connecting DFC to these traditional measures (i.e. program length, time complexity and circuit size complexity) we support the contention that DFC is a reflection of a very general property, as we would hope would be the case for a measure of pattern-ness.

²By Frank Brown of AFIT.

4.2 Decomposed Function Cardinality

We need a more formal notation for the representation of a function's decomposition to define Decomposed Function Cardinality (*DFC*). For $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$, a representation r_f of f is a finite, acyclic, directed graph G and a set of finite functions P . That is, $r_f = (G, P)$. G and P are defined below.

It will simplify the notation if we rename some sets. We will use U_i to represent input (X_j), intermediate (Z_k), and output (Y) sets. In particular, let

$$U_i = \begin{cases} X_i & i = 1, 2, \dots, n \\ Z_{i-n} & i = n+1, n+2, \dots, n+[P]-1 \\ Y & i = n+[P]. \end{cases}$$

In this notation,

$$f : U_1 \times U_2 \times \dots \times U_n \rightarrow U_{n+[P]},$$

and $U_{n+1}, U_{n+2}, \dots, U_{n+[P]-1}$ are the intermediate ("feature") sets that are created as a result of the decomposition.

The graph (G) consists of a set of vertices (V) and a set of arcs (A). That is, $G = (V, A)$. There is a vertex in the graph for each variable in the representation, i.e. $V = \{u_1, u_2, \dots, u_{n+[P]}\}$. The lower case u_i is a variable for the set U_i . A is a subset of $V \times V$, called a set of arcs. Indegree of u_i is 0 for $i = 1, 2, \dots, n$; outdegree of $u_{n+[P]}$ is 0; indegree of $u_{n+[P]}$ is 1; indegree and outdegree of u_i are not 0 for $i = n+1, n+2, \dots, n+[P]-1$.

$P = \{p_1, p_2, \dots, p_{[P]}\}$ is a set of nonempty and nonconstant functions of the following form:

$$p_j : \prod_{i \in I_j} U_i \rightarrow U_j, \quad j = n+1, n+2, \dots, n+[P],$$

where I_j is a set of the indices of the input variables for p_j . That is, with I the set of positive integers,

$$I_j = \{i \in I \mid (u_i, u_j) \in A\}.$$

A and P are such that $(u_i, u_j) \in A$ if and only if there exists a $p_j \in P$ which has $i \in I_j$. Let n_j be the number of variables input to p_j . That is, $n_j = |I_j|$.

In summary,

$$r_f = (G, P),$$

$$G = (V, A),$$

$$V = \{u_1, u_2, \dots, u_{n+[P]}\},$$

$$A \subseteq V \times V,$$

$$P = \{p_1, p_2, \dots, p_{[P]}\},$$

and

$$p_j : \prod_{i \in I_j} U_i \rightarrow U_j.$$

DFC (c.f. [48, pp.37-48]³) is used with two meanings. First, it is used to denote the *DFC* of a particular representation of function,

$$DFC(r_f) = \sum_{j=1}^{[P]} (\prod_{i \in I_j} [U_i]).$$

When $U_j = \{0, 1\}$ for all j , this becomes:

$$DFC(r_f) = \sum_{j=1}^{[P]} 2^{n_j}.$$

We are occasionally interested in using the actual cardinality $[p_j]$ of the component functions rather than 2^{n_j} . We call this Decomposed Partial Function Cardinality (DPFC),

$$DPFC(r_f) = \sum_{j=1}^{[P]} [p_j].$$

When all the component functions of a representation are total the two measures *DFC* and *DPFC* are the same.

When we talk about the *DFC* of a function, we mean the *DFC* of that function's optimal representation. If \mathcal{R} is the set of all possible representations then

$$DFC(f) = \text{minimum over } r_f \in \mathcal{R} \text{ of } (DFC(r_f)).$$

The DPFC of a function is similarly defined. We also use simply *DFC* (or DPFC) when the particular function or representation is made clear by the context.

4.3 Decompositions Encoded as Programs

4.3.1 Introduction

We want to relate DFC to program length as developed in Appendix A. In particular, we want to formalize how representations of decompositions are special cases of programs. The cost function on representations that then corresponds to program length is of special interest since we know a number of properties of program length. We must also address the concern that a decomposition of a function consists of both component functions and their interconnections while DFC only measures the complexity of the component functions. After all, it could be the case that the complexity of the interconnections is independently important in the true measure of pattern-ness.

We will be concerned with a binary function $f: \{0, 1\}^n \rightarrow \{0, 1\}$ and representations of such functions denoted r_f . We will also be interested in the optimal representation

³In the notation of [48], *DFC* is exactly $f_c(r_{f-opt})$ when $w(r) = 1$ for all r in R' and R' is the set of all Boolean functions of the form $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$, m and n positive integers.

of a given function f which will be denoted R_f . Let A^* denote the set of all finite strings from alphabet A . Define an encoding procedure $e : \mathcal{R} \rightarrow \{0,1\}^*$, where \mathcal{R} is the set of all possible representations. We prove that $e(r_f)$ is a program in the formal sense of Appendix A. We define a cost function on encodings as simply their string length, $L(e(r_f))$. L is like a combinational form of Kolmogorov complexity and has many of its properties (see [33] and [54, Section 5.7]). We use the relationships of Appendix A to prove a number of properties of e . In particular we prove that for all f

$$2(n+1) \leq L(e(R_f)) \leq 2^n + 1,$$

there exists a function f such that

$$L(e(R_f)) = 2^n + 1,$$

and that

$$2^n \leq \text{average}(L(e(R_f))) < 2^n + 1.$$

With respect to the concern that DFC does not reflect the interconnection complexity, we prove that if DFC is small then $L(e(R_f))$ (which does reflect interconnection complexity) is also small.

4.3.2 Encoding Procedure

We now define a method for encoding decompositions into binary strings. The encoding procedure produces a binary string $e(r_f)$ which is an identifier of a unique function of the form $f : \{0,1\}^n \rightarrow \{0,1\}$. The procedure for generating encodings is defined on \mathcal{R}' , a subset of \mathcal{R} . In particular, using the notation of Section 4.2, \mathcal{R}' is the subset of \mathcal{R} such that:

1. $[P] \leq 2^n$,
2. $[p_i] \leq 2^n$ for $i = 1, 2, \dots, [P]$,
3. $[A] \leq 2^n$.

We will prove that although \mathcal{R}' is a proper subset of \mathcal{R} , \mathcal{R}' includes all optimal representations. We use the notation $\lceil a \rceil$ to specify the smallest integer that is greater than or equal to the real number a . Similarly, $\lfloor a \rfloor$ is the largest integer less than or equal to a . There is one other relationship which is a direct consequence of 1 above.

Theorem 4.1 *For any r_f in \mathcal{R}' an arc in A can be specified with n^2 bits for $n \geq 4$.*

Proof:

By definition of V , $[V] \leq (n + [P])$. Thus, a v_i in V can be specified with $\lceil \log(n + [P]) \rceil$ bits (log to the base 2). By constraint 1 on \mathcal{R}' , $\lceil \log(n + [P]) \rceil \leq \lceil \log(n + 2^n) \rceil \leq \lceil n \log(n) \rceil$. Finally, since an arc can be defined by its head and tail vertices, an arc can be specified with $2\lceil n \log(n) \rceil$ bits, which is less than or equal to n^2 for $n \geq 4$.

□

The objective of the following encoding scheme is to encode any reasonable r_f into as short a binary string as possible while keeping a manageable expression for $L(e(r_f))$. We assume that n is known. If n is not known, a unary representation of n using $n + 1$ bits could be added to the front of the encoding. The encoding assumes that all functions involved are total. If a partial function is involved, it can be made total by arbitrarily assigning all Don't-Cares to be 0. The encoding procedure is as follows:

1. The first bit of $e(r_f)$ indicates whether or not the function is decomposed. When this bit is 0, the function is not decomposed and the rest of the program lists, in the order of the domain of f , all the images of f . When this bit is 1 the rest of the encoding is as follows.
2. The next n bits specify $[P]$, which is possible by the first constraint on \mathcal{R}' .
3. If $[P]$ is zero then f is either a constant or a projection of one of its variables. When $[P]$ is zero the next $n + 1$ bits of the encoding indicate which constant or projection function. If f is a projection of the i^{th} variable then the i^{th} of the first n bits is one and the others are zero. If f is a constant, which will be indicated by the first n bits being all zero, then the $n + 1^{th}$ bit indicates which constant function is f . The total encoding of any constant or projection function therefore requires $2(n + 1)$ bits. If f is not a constant or projection function then $[P]$ is not zero and the encoding proceeds as follows.
4. The encoding repeats the following for $i = 1 \dots [P]$
 - a. The first n bits specify $[p_i]$, which is possible by the second constraint on \mathcal{R}' .
 - b. The next $[p_i]$ bits specify p_i .
5. The next n bits of the program specify $[A]$, which is possible by the third constraint on \mathcal{R}' .
6. The encoding then repeats the following for $i = 1 \dots [A]$. The i^{th} arc is specified by n^2 bits, which is possible by Theorem 4.1.

4.3.3 Length of an Encoding

We now develop the expression for the length of the string which results from the encoding procedure.

Theorem 4.2 *If $[P] = 0$ then $L(e(r_f)) = 2(n + 1)$; if $[P] = 1$ then $L(e(r_f)) \leq 1 + 2^n$; otherwise $L(e(r_f)) = 1 + 2n + n^2[A] + n[P] + DFC(f)$.*

Proof:

When $[P] = 0$ there is the bit of step 1, the n bits of step 2 and the $n + 1$ bits of step 3. When $[P] = 1$, the worst case is when there are no vacuous variables and then there is the bit of step 1 and the 2^n bits of f . Otherwise: Step 1 uses 1 bit. Step 2 uses n bits. Step 4 uses a sum as $i = 1, 2, \dots, [P]$ of the bits required for part a) plus the bits required for part b). Part a) requires n bits. Part b) requires $[p_i]$ bits. The total number of bits required for Step 4 then is:

$$\sum_{i=1}^{[P]} (n + [p_i]) = n[P] + \sum_{i=1}^{[P]} [p_i] = n[P] + DFC(f).$$

Step 5 uses n bits. Step 6 uses a sum as $i = 1$ to $[A]$ of the bits required for an arc. An arc requires n^2 bits, assuming $n \geq 4$. The total number of bits required for Step 6 then is:

$$\sum_{i=1}^{[A]} n^2 = n^2[A].$$

Therefore,

$$L(e(r_f)) = 1 + n + n[P] + DFC(f) + n + n^2[A] = 1 + 2n + n^2[A] + n[P] + DFC(f).$$

□

4.3.4 \mathcal{R}' Includes All Optimal Representations

As mentioned earlier, \mathcal{R}' is a proper subset of \mathcal{R} ; for example, we could define an r_f with arbitrarily many identity functions so $[P]$ would be larger than that allowed for encoding. However, \mathcal{R}' includes most reasonable representations and we prove that \mathcal{R}' includes all "optimal" representations. For every f , define r_{f-opt} as a representation of f such that $L(e(r_{f-opt})) \leq L(e(r_f))$ for all r_f that represent f . In order to prove that all optimal representations are in \mathcal{R}' we need several results which we develop now.

Theorem 4.3 For $n \geq 3$, $2(n + 1) \leq L(e(r_{f-opt})) \leq 1 + 2^n$.

Proof:

The right inequality follows since an arbitrary f has a representation with $[P] = 1$ (i.e. $p = f$) and whose length is $1 + 2^n$. The left inequality follows immediately if r_f is not a decomposition. If r_f is a decomposition then either, $L(e(r_f)) = 1 + 2n + n^2[A] + n[P] + DFC(f) \geq 2(n + 1)$ or $L(e(r_f)) = 2(n + 1)$.

□

The DFC of a total function cannot be greater than its cardinality.

Theorem 4.4 $DFC(f) \leq 2^n$.

Proof:

From Theorem 4.3, $L(e(r_{f-opt})) \leq 1 + 2^n$. By Theorem 4.2, $1 + 2n + n^2[A] + n[P] + DFC(f) \leq 1 + 2^n$ or $2n + n^2[A] + n[P] + DFC(f) \leq 2^n$. Since all the terms on the left are positive, each term is less than or equal to 2^n . In particular, $DFC(f) \leq 2^n$.

□

No component of a decomposition can be larger than the whole DFC.

Theorem 4.5 $[p_i] \leq 2^n$ for $i = 1, 2, \dots, [P]$.

Proof:

From Theorem 4.4, $DFC(f) \leq 2^n$. Since all the $[p_i]$ terms that sum to $DFC(f)$ are positive, each term must be less than or equal to 2^n .

□

The number of components cannot exceed the DFC.

Theorem 4.6 $[P] \leq 2^n$.

Proof:

$n[P]$ is a term in $L(e(r_{f-opt}))$ by Theorem 4.2 and must be less than or equal to 2^n by Theorem 4.3. Finally, since $n \geq 1$ we have $[P] \leq 2^n$.

□

The number of arcs cannot exceed the DFC.

Theorem 4.7 $[A] \leq 2^n$.

Proof:

Follows as in Theorem 4.6.

□

Finally we can prove that \mathcal{R}' includes all optimal representations.

Theorem 4.8 For all f of the form $f : \{0,1\}^n \rightarrow \{0,1\}$, $r_{f-opt} \in \mathcal{R}'$.

Proof:

Follows from Theorems 4.5, 4.6, and 4.7 and the definition of \mathcal{R}' .

□

4.3.5 Properties of Encodings

We now can relate the encoded representations to the programs of Appendix A.

Theorem 4.9 $e(r_f)$ is a program.

Proof:

The set of $e(r_f)$ for all $r_f \in \mathcal{R}'$ is a language P satisfying the prefix condition. For F as the set of all functions of the form $f : \{0, 1\}^n \rightarrow \{0, 1\}$ define $M : P \rightarrow F$ such that $M(e(r_f)) = f$. Under these conditions, (P, F, M) form a programmable machine as defined in Appendix A.

□

Because $e(r_f)$ is a program, all the results about programs apply to $e(r_f)$. We want to highlight a few of these results in the present context.

Theorem 4.10 *There exists a function f such that $L(e(r_{f-opt})) = 1 + 2^n$.*

Proof:

Suppose to the contrary that no such function existed. That is, the worst cost of any function is 2^n or less. We know there exist functions with cost strictly less than 2^n , e.g. a constant function. In this situation the average $L(e(r_f))$ is strictly less than 2^n ; that is, we have an average of a set of finite numbers containing some numbers less than 2^n but containing no numbers greater than 2^n . However, since $e(r_f)$ is a program the average $L(e(r_f))$ being strictly less than 2^n contradicts Theorem A.16. Therefore the supposition is false and the theorem is proven.

□

Theorem 4.11 $2^n \leq \text{average } L(e(r_{f-opt})) < 1 + 2^n$.

Proof:

The left inequality follows from Corollary A.3. For the right inequality, we know that there exist functions with cost strictly less than $2^n + 1$, e.g. a constant function, and that there are no functions with cost greater than $2^n + 1$ by Theorem 4.3. Therefore the average $L(e(r_f))$ is strictly less than $2^n + 1$.

□

4.3.6 Decomposed Function Cardinality and Program Length

We are concerned with the question: "How well does $DFC(f)$ capture the essential complexity of a function?" In Appendix A we developed the idea of program length as a very general characterization of size complexity. In this section we found that $e(r_f)$ is a program with length $L(e(r_f)) = 1 + 2n + n^2[A] + n[P] + DFC(f)$. Therefore, one step in relating $DFC(f)$ to general complexity is to assess its role in $L(e(r_f))$.

In those cases where $L(e(r_{f-opt})) = 1 + 2^n$, $DFC(f) = [f] = 2^n = L(e(r_{f-opt})) - 1$. That is, in most cases $DFC(f)$ is almost exactly $L(e(r_{f-opt}))$. When $L(e(r_{f-opt})) < 1 + 2^n$, we do not have as simple a relationship. However, we can prove that $DFC(f)$ is roughly of the same order of complexity as $L(e(r_{f-opt}))$.

We already know that $L(e(r_{f-opt}))$ cannot be small unless $DFC(f)$ is small since $DFC(f)$ is a part of $L(e(r_{f-opt}))$. Our concern is that $DFC(f)$ might be small while $L(e(r_{f-opt}))$ is large. That is, we do not want to think that a function is patterned based on DFC while it really is not patterned when you consider the full cost of representation as measured by $L(e(r_{f-opt}))$. In order to demonstrate that this is not a problem, we will show that $L(e(r_{f-opt})) \leq n^3 DFC(f)$. That is, $L(e(r_{f-opt}))$ is never of a much higher order than $DFC(f)$.⁴

Theorem 4.12 $DFC(f) < L(e(r_{f-opt})) \leq n^3 DFC(f)$ for all $n \geq 4$ and $DFC(f) \geq 1$.

Proof:

The left inequality follows immediately. For the right, let n_i be the number of input variables for p_i . Then $DFC(f) = \sum_{i=1}^{[P]} 2^{n_i}$. Since $n_i \geq 2$ we have $DFC(f) \geq 4[P]$. Also, $[A] = \sum_{i=1}^{[P]} n_i + 1$ and since $n_i \leq n$ we have $[A] \leq n[P] + 1$. Now use the second inequality to eliminate $[A]$ from the expression for $L(e(r_{f-opt}))$ to get:

$$L(e(r_{f-opt})) \leq 1 + 2n + n^2(n[P] + 1) + n[P] + DFC(f)$$

and then use the first inequality to eliminate $[P]$ to get:

$$L(e(r_{f-opt})) \leq 1 + 2n + n^2(n\frac{1}{4}DFC(f) + 1) + n\frac{1}{4}DFC(f) + DFC(f)$$

or equivalently:

$$L(e(r_{f-opt})) \leq 1 + 2n + n^2 + \frac{1}{4}DFC(f)(n^3 + n + 4)$$

This simplifies to $L(e(r_{f-opt})) \leq n^3 DFC(f)$ for $n \geq 4$ and $DFC(f) \geq 1$.

□

From Theorem 4.12 we know that if $DFC(f)$ is small then program length is also small. For example, if $DFC(f)$ is polynomial in n then program length is also polynomial in n . The point is that $DFC(f)$ reflects the essential complexity even though it does not directly include a measure of the interconnection complexity of the representation.

There is another indication of the relative importance of interconnection complexity versus DFC. If the interconnections are minimized without regard to DFC then the result is $n + 1$ interconnections (n the number of non-vacuous variables). That is, all functions on a given number of variables have the same minimum interconnection complexity.

⁴c.f. [54] Theorem 5.7.5.

4.4 Decomposed Function Cardinality and Time Complexity

Another concern about the $DFC(f)$ measure of pattern-ness is that it is based on combinational machines. What if there are patterns that only become measurable with respect to sequential machines? If this were the case then $DFC(f)$ would not measure the patterns of interest in most real computing problems. Of course, this is not the case. In order to demonstrate that $DFC(f)$ will be small whenever there exists a good sequential algorithm, we relate $DFC(f)$ to time complexity (the traditional measure of Computational Complexity Theory). There are several differences in the two perspectives (time complexity versus $DFC(f)$) that must be considered. One difference is that $DFC(f)$ considers complexity in terms of a single measure of a whole finite function while traditional complexity is in terms of a particular input to an infinite function. A second difference is that Pattern Theory is based on size complexity while traditional measures are based on time complexity. The objective of this section is to demonstrate that size and time complexity are simply different perspectives of what might be considered the essential computational complexity of a function.

Why not just use time complexity in the first place? First and foremost, the time complexity of all finite functions is $O(1)$. Therefore, time complexity does not differentiate between patterned and un-patterned finite functions. Another problem with time complexity is finding out what it is. We know how to find the time complexity of an algorithm. But we do not know whether or not that is the time complexity of the function. It may simply be a poor algorithm. This also presupposes that you have an algorithm for the function, which is begging our question.

Once the relationship between $DFC(f)$ and traditional time complexity is established we are able to apply results from the traditional theory of computational complexity. Also, because small time complexity implies small $DFC(f)$ we know that patterns as measured by time complexity are a subset of patterns as measured by $DFC(f)$.

Traditional time complexity $t(n)$ is defined in terms of the run-time of a Turing Machine [59] which realizes a function of the form $f : \{0,1\}^* \rightarrow \{0,1\}$, when the input is a string of length n . On the other hand, $DFC(f)$ is defined as a measure of a realization of a function of the form $g : \{0,1\}^n \rightarrow \{0,1\}$. In order to be able to compare these two measures we use the following device. Let $f : \{0,1\}^* \rightarrow \{0,1\}$ be a function and $f_n : \{0,1\}^n \rightarrow \{0,1\}$ for $n = 0, 1, 2, \dots$ be a sequence of functions such that $f_n(x) = f(x)$ for all x in $\{0,1\}^n$ and for all n in N . The DFC complexity of a representation of f_n is denoted $s(n)$.

We developed this device and the following theorem not knowing that a similar result had been previously demonstrated (see [19, 55]). However, because this is fundamentally important to Pattern Theory, we want to present a proof. Rather than repeating the original proof, we present our proof and suggest that the reader see [64, pp.271-276] for a more elegant demonstration of this result.

Theorem 4.13 *If $f : \{0,1\}^* \rightarrow \{0,1\}$ is a function with an algorithm of worst-case time complexity $t(n)$ then for every $n \in N$, there exists a combinational realization of f_n such that $s(n)$ is in $O(t(n) \log t(n))$.*

Proof:

Let $M = \{\Sigma, \Gamma, Q, q_0, \delta, F\}$ be the Turing Machine which realizes f with worst case time complexity $t(n)$, where Γ is the tape alphabet, Σ is a subset of Γ called the input alphabet, Q is a set of machine states, q_0 is the start state, F is a set of final states, and $\delta : \Gamma \times Q \rightarrow \Gamma \times Q \times \{-1, +1\}$ is the transition function (reference [59, pp.211-215]). The L (for left) and R (for right) of Sudkamp's definition have been replaced with -1 and $+1$; the reason for this will be pointed out later. We break δ into $\delta' : \Gamma \times Q \rightarrow \Gamma$, $\delta'' : \Gamma \times Q \rightarrow Q$, and $\delta''' : \Gamma \times Q \rightarrow \{-1, +1\}$ such that $\delta(g, q)$ equals $(\delta'(g, q), \delta''(g, q), \delta'''(g, q))$.

Define the interval $Z = \{0, 1, 2, \dots, t(n)\}$ and the function $p : Z \rightarrow Z$ such that $p(i)$ is the position of the tape after i transitions. Define $q : Z \rightarrow Q$ such that $q(i)$ is the state of the machine after i transitions. Define $g : Z \times Z \rightarrow \Gamma$ such that $g(i, j)$ is the symbol in the j^{th} position of the tape after i transitions. We use p, q, g and h to establish a combinational realization of f_n . Define $h : \Gamma^{t(n)} \times Z \times Q \rightarrow \Gamma^{t(n)}$ such that $g(i, j) = h(g(i-1, j), p(i-1), q(i-1))$ for all i and j .

From the starting conditions of the Turing Machine $p(0) = 1, q(0) = q_0$, and $g(0, j) = x(j)$ (i.e. the input) for $j = 1, 2, \dots, n$. From the final conditions, $p(t(n)) = 0$ and $q(t(n))$ is the state defined to correspond to $f(x) = 0$ and $f(x) = 1$ as appropriate.

By the definition of a Turing Machine we can write difference equations for p, q , and g :

$p(i) = p(i-1) + \delta'''(g(i-1, p(i-1)), q(i-1))$ for $i = 1, 2, \dots, t(n)$. Using $\{-1, +1\}$ as the range of δ''' , rather than $\{L, R\}$, allows this simple addition. The increment in cost associated with each i is the cost of addition plus the cost of δ''' , i.e. $s(+) + s(\delta''')$, since the cost of all the other functions are accounted for elsewhere. The cost of "+" is $O(m)$ (reference [7] where, since the largest number to be added is $t(n)$, $m = \log t(n)$). That is, $s(+) = k \log t(n)$. For $\delta''' : \Gamma \times Q \rightarrow \{-1, +1\}$, we have $s(\delta''') = [\Gamma][Q] \log\{\{-1, +1\}\} = [\Gamma][Q] = k$, k some constant. Therefore, $s(p) = k_1 \log(t(n)) + k_2$.

$q(i) = \delta''(g(i-1, p(i-1)), q(i-1))$, for $i = 1, 2, \dots, t(n)$. The increment in cost associated with each i is $s(\delta'')$, since the cost of all the other functions are accounted for elsewhere. For $\delta'' : \Gamma \times Q \rightarrow Q$, we have $s(\delta'') = [\Gamma][Q] \log[Q] = k$, k some constant. Therefore, $s(q) = k$.

$g(i, j)$ is unchanged for all j except $j = p(i-1)$ and $g(i, p(i-1)) = \delta'(g(i-1, p(i-1)), q(i-1))$. That is, $g(i, j) = h(g(i-1, j), p(i-1), q(i-1)) = .NOT.(.EQ.(j, p(i-1))) \times g(i-1, j) + .EQ.(j, p(i-1)) \times \delta'(g(i-1, j), q(i-1))$, where the functions $.NOT.$ and $.EQ.$ are the obvious logical operators with values 0 or 1 and $+$ and \times are arithmetic operators. The increment in cost associated with h is $s(.NOT.) + s(+) + s(\delta') + 2s(.EQ.) + 2s(\times)$. The multiplications always involve a zero or a one, therefore we assume that $s(\times)$ is constant. The addition always involves a zero, therefore we assume $s(+) is constant. The cost of the logical operators is constant.$

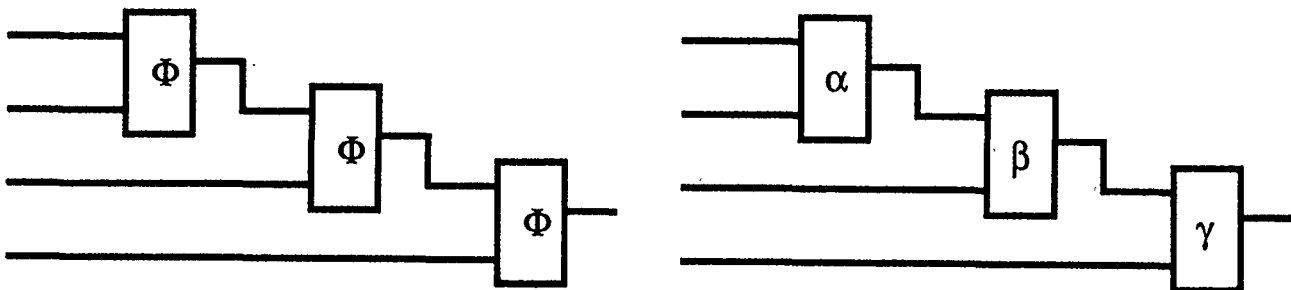


Figure 4.7: Similar Decompositions, One Recursive, One Not

For $\delta' : \Gamma \times Q \rightarrow \Gamma$, we have $s(\delta') = [\Gamma][Q] \log[\Gamma]$ which is again some constant. Thus, $s(h) = k$, for some constant k . There must be an h for each tape position. A maximum of $t(n)$ tape positions will be used, therefore, we know there can be less than or equal to $t(n)$ h 's. The total cost then for updating $g(i, j)$ for all j at each i is $kt(n)$.

Each transition can be combinationaly realized with a cost of $s(p) + s(q) + s(h) = k_1 t(n) + k_2 + k_3 \log(t(n))$. The entire process can be combinationaly realized by modeling $t(n)$ transitions. Inputs which require less than $t(n)$ transition can be dealt with by modifying δ to include a no-op type transition. Therefore, the total combinational complexity is $t(n)\{k_1 \log(t(n)) + k_2 + k_3 \log(t(n))\}$. That is, $s(n)$ is in $O(t(n) \log(t(n)))$.

□

An immediate consequence of this Theorem is that if it is not possible to find a good (relative to *DFC*) combinational realization of a function then it is also not possible to find a good (relative to time complexity) algorithm for the function. In other words, if a method for finding optimal combinational realizations fails to find a nice representation of a function then there does not exist a nice algorithm to compute that function either.

The results in computability theory demonstrate that recursion is the key property that a function must have to be computable. It is tempting then to extend this and say that recursion is a key property that a function must have to be patterned (i.e. not complex). However, *DFC*(f) does not favor functions with recursive representations. For example, Figure 4.7 shows two functions of equal *DFC*. The function on the left has a recursive-like representation while the function on the right does not. We feel that the problem lies in trying to extend computability results to complexity

rather than in $DFC(f)$. Recursion is important in getting finite representations of infinite functions (see Appendix A); however, it is not crucial to the complexity of finite functions. There is a real cost savings in recursively reusing components of a composition; however, we believe that this is a secondary effect and it is not indicative of the existence or absence of patterns. It would be easy to redefine DFC to reflect the economy of reusing functions in a composition; for example, we could include only the unique p_i 's in adding up DFC. However, finding this newly defined DFC would be much more difficult. We feel that this improvement in generality is not worth the loss in tractability.

In summary, it has been proven that if a function has low time complexity then it also has low DFC. This means that if a function has a nice representation using sequential constructs (do-while, recursion, etc.) then it also has a nice combinational representation. The main point is that there is no loss in generality due to the PT 1 restriction to combinational machines.

4.5 Decomposed Function Cardinality and Circuit Complexity

There has been a great deal of theoretical work on measuring and minimizing the complexity of electronic circuits. Using Savage'76 [54] as the principal reference, we review the several measures of complexity. "Combinational" complexity is the number of circuit elements required to realize a function. "Formula size" is the number of operators in an algebraic expression of the function. "Delay complexity" is the length of the longest path from input to output in a realization of a function. Combinational complexity is very similar to Decomposed Function Cardinality (DFC). We will discuss their relationship first. The relationship of combinational complexity to formula size and depth complexity is then summarized from Savage'76.

Combinational complexity in Savage'76 is defined relative to a set of basis functions (Ω). A given Boolean function f is then realized by combinations of the elements of Ω . The combinational complexity $C_\Omega(f)$ of a Boolean function f relative to the basis Ω is the minimum number of elements from Ω required to realize f .

$C_\Omega(f)$ is sometimes generalized by allowing the various elements of Ω to have different costs [52]. For example, we could define a weighting function ($\omega : \Omega \rightarrow \mathcal{R}$, \mathcal{R} the real numbers) on Ω . Then the generalized cost of a function ($C_{\Omega,\omega}(f)$) is the sum of the weights of the elements in the realization that minimizes that sum. $C_\Omega(f)$ is the special case of $C_{\Omega,\omega}(f)$ where ω is the constant 1 function.

DFC is also a special case of this generalized combinational complexity. DFC is exactly $C_{\Omega,\omega}(f)$ when Ω is the set of all Boolean functions and $\omega(p) = [p]$ for all $p \in \Omega$.

Note that for the most common set of basis functions, $\Omega = \{AND, OR, NOT\}$, $C_\Omega(f)$ and DFC are also very similar.

Theorem 4.14 *For a Boolean function f and basis set $\Omega = \{AND, OR, NOT\}$, $DFC(f) \leq 4C_\Omega(f) \leq 4nDFC(f)$.*

Proof:

$DFC \leq 4C_\Omega$ since each element of Ω has cardinality of 4 or less. Suppose the decomposition of f with minimum DFC is made up of p_1, p_2, \dots, p_P and the number of variables going into p_i is n_i . In this case, $DFC(f) = \sum_{i=1}^P 2^{n_i}$. Now, $C_\Omega(f) \leq \sum_{i=1}^P C_\Omega(p_i)$. Since each p_i has a sum-of-products representation, we have $C_\Omega(p_i) \leq n_i 2^{n_i}$ [54, p.19]. Therefore, $C_\Omega(f) \leq \sum_{i=1}^P n_i 2^{n_i} \leq n \sum_{i=1}^P 2^{n_i} = n DFC(f)$.

□

In summary, C_Ω is in some ways more general than DFC; but, it is more general in a way that denies any absolute meaning to complexity. That is, for all functions f there exist an Ω (namely any Ω with f as an element) such that $C_\Omega(f) = 1$. This relativity of complexity to a chosen basis is believed by some to be unavoidable. The main idea of Pattern Theory is that there is some general absolute measure of complexity in the sense of patterns.

$C_{\{AND, OR, NOT\}}$ is quantitatively similar to DFC but it leads you to artificially decompose (i.e. represent in some normal form) un-patterned functions. In Pattern Theory, all functions have themselves as a normal form representation. Also, on smaller functions it gives an artificial importance to members of Ω .

Savage'76 has little to say about the relationship of combinational complexity to formula size and depth complexity. Depth complexity is proportional to the logarithm of C_Ω . Also, $C_\Omega \leq$ formula size. Other than that, not much is known about the relationship between combinational complexity, depth complexity and formula size.

4.6 Summary

This chapter defines our chosen figure-of-merit for algorithm design: Decomposed Function Cardinality (DFC). We then support the idea that DFC reflects the essential pattern-ness of a function. Chapter 6 has the results of many experiments that support this idea. In this chapter we supported this idea by showing that if a function has an interesting pattern by most other measures then it also has a pattern according to DFC. We considered three other measures of computational complexity: program length, time complexity and circuit complexity.

Appendix A interprets a result from Communications Theory in terms of programs and the length of a program (essentially the number of characters in a listing of a program). It turns out that a similar interpretation had previously been made [12]. We then related DFC and program length. This relation supports the use of DFC rather than program length as our measure of patterns because DFC is more tractable and yet reflects the essential complexity that program length would measure.

We developed a formal proof of the relationship between DFC and time complexity. It turns out that this relationship had also been previously demonstrated [54]. The relationship between DFC and time complexity demonstrates that DFC is a more general measure of complexity (that is, anything patterned relative to time complexity will also be patterned relative to DFC). DFC also has the advantage over time

complexity in that it is meaningful for finite functions and allows for a method of design (Chapter 5).

The relationship between DFC and circuit complexity is quite simple. DFC is a special case of the more general definition of circuit complexity. We believe it is the special case that reflects the essential pattern-ness of a function.

We mentioned earlier that it would be desirable to include the cost of the interconnections in a general measure of patterns, as in the program length of an encoding of a decomposition. It would also be desirable for a general measure to reflect the savings of reusing components, as in a recursive representation. However, we believe that DFC allows you to determine the basic degree of pattern-ness without the complexity of dealing with these secondary effects.

Chapter 5

Function Decomposition

5.1 Introduction

By a "function" we mean the traditional mathematical function; that is, a function is an association between inputs and outputs such that for every input x there is exactly one output $f(x)$. Functions in general may have several inputs, e.g. $f(x, y, z)$. The decomposition of a function is an expression of that function in terms of a composition of other (usually smaller) functions. For example, if $f(x_1, x_2, \dots, x_8) = F[\lambda[\theta(x_1, x_2), \phi(x_3, x_4), \psi(x_5, x_6)], \zeta(x_7, x_8)]$ then the right-hand side of the equation is a decomposition of the function f . The process of finding a decomposition of a function is called function decomposition.

Function decomposition is of practical importance in the design of computational systems. The realization of a large function in terms of smaller functions has a number of practical benefits, especially simplifying the design process and lowering the cost of the overall realization. The development of function decomposition theory has been motivated primarily by Switching (or Logic) Circuit Design, where the lowest level sub-functions are realized by some standard circuit component (e.g. an AND gate). Pattern Theory holds that function decomposition is of fundamental importance in the development of all computing systems, including algorithm design and machine learning. Aside from the practical motivation, function decomposition is a well defined and very interesting mathematical problem.

This chapter has three main sections. Section 5.2 introduces and formally proves the test for decomposability. The next section describes the Ada program used in this project to decompose functions. The final section reports on performance tests of various versions of the Ada program.

5.2 The Basic Decomposition Condition

5.2.1 Introduction

A number of algorithms have been developed to find the decompositions of a given function. All these algorithms are iterative. That is, they decompose the function in a series of similar steps. The first step decomposes the function into a small number of sub-functions. The second step decomposes each of these sub-functions into a small number of sub-subfunctions. This process is repeated until the remaining functions are no longer decomposable. The function decomposition algorithms use a fairly standard test for whether or not a function (or sub-function) decomposes. This test is based on what we call the *basic decomposition condition*. On the other hand, the function decomposition algorithms use different methods for searching and selecting among possible decompositions. An optimal method of search and selection (other than exhaustive search) has not been defined. The basic decomposition is the common non-heuristic portion of these algorithms. The basic decomposition condition test is an exponentially (relative to the number of variables in the function) difficult problem. The number of variable partitions is another exponential factor, however, this factor can perhaps be mitigated with reasonable search heuristics. The nature of the decomposition test is such that there is no way to limit its exponential complexity. Therefore, the decomposition test is also important as a limiting factor in the practicality of any algorithm which exactly decomposes total functions.

The purpose of this section is to develop a concise, yet rigorous statement of a general basic decomposition condition. Many of the previously published statements of the basic decomposition condition have shortcomings in rigor, generality, or in requiring an extensive background in non-essential materials. For example, the bible of function decomposition is the text by H. A. Curtis [14]. However, its not until page 471 that the most general form of the decomposition condition is given and an understanding of much of the previous 470 pages is required to understand page 471. Reference [14] also does not prove the most general statement of the decomposition condition, rather it is an extension of the proofs of less general forms. The most general form given in [14] is also not applicable to multi-valued functions. Finally, [14] has been out of print for some time and it is very difficult to find.

5.2.2 An Intuitive Introduction to the Decomposition Condition

Consider a function of the form $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$. This function is also denoted $f(x_1, x_2, \dots, x_n)$, where x_i represents some unspecified value of X_i . $\{x_1, x_2, \dots, x_n\}$ is called the set of variables of f . We are interested in a partition of the variables of f into two sets. A partition is a collection of subsets whose union is the whole set and whose intersections are all empty. We denote the two sets of variables v_1 and v_2 . Therefore, $v_1 \cup v_2 = \{x_1, x_2, \dots, x_n\}$ and $v_1 \cap v_2 = \emptyset$. For example, if

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Table 5.1: A Table Representation of a Function

$n = 8$, then one partition is $(v_1 = \{x_1, x_3, x_4, x_6, x_7\}, v_2 = \{x_2, x_5, x_8\})$ and a second partition is $(v_1 = \{x_2, x_3, x_4, x_6, x_7\}, v_2 = \{x_1, x_5, x_8\})$.

A finite function can be represented by a "truth-table" where all possible values for the input are listed with their corresponding image under the function. Table 5.1 is a table defining a function.

We might call Table 5.1 a one-dimensional table since it lists the values in a vertical line. We can also represent a function with a two-dimensional table by letting the values of the variables in v_1 mark off one direction while the values of the variables in v_2 mark off the orthogonal direction. The value of the function, for the given values of v_1 and v_2 , then go into the "matrix" (the 2-D table) at the coordinates (v_1, v_2) . For example, if $v_1 = \{x_1, x_2\}$ and $v_2 = \{x_3, x_4\}$ then the function of Table 5.1 could also be represented as the 2-D table of Table 5.2.

In Table 5.2, x_1 and x_2 specify a column of the 2-D table while x_3 and x_4 specify a row of the table. Of course, x_1, x_2, x_3 , and x_4 specify a particular point in the table corresponding to the value of the function at x_1, x_2, x_3 , and x_4 . A 2-D table of a function is called a *partition matrix*. A different partition of the variables would give a different arrangement of f 's values. For example, with respect to the partition $v_1 = \{x_1, x_4\}$ and $v_2 = \{x_2, x_3\}$ the 2-D table becomes as in Table 5.3.

There could also be partitions with unequal numbers of variables as in Table 5.4. If one of the sets of variables is empty we have the familiar 1-D table as in Table 5.1.

Now examine Table 5.5. Notice that all the columns are the same. It does not

		x_1x_2			
x_3x_4	f	00	01	10	11
	00	0	1	1	1
	01	1	0	0	0
	10	1	1	0	1
	11	0	1	0	1

Table 5.2: A 2-D Table of a Function With Respect to a Partition of its Variables

		x_1x_4			
x_2x_3	f	00	01	10	11
	00	0	1	1	0
	01	1	0	0	0
	10	1	0	1	0
	11	1	1	1	1

Table 5.3: A Second 2-D Table of a Function With Respect to a Partition of its Variables

		x_1	
$x_2x_3x_4$	f	0	1
	000	0	1
	001	1	0
	010	1	0
	011	0	0
	100	1	1
	101	0	0
	110	1	1
	111	1	1

Table 5.4: A Table Representation of a Function

		x_1x_2			
f		00	01	10	11
x_3x_4	00	1	1	1	1
	01	0	0	0	0
	10	0	0	0	0
	11	1	1	1	1

Table 5.5: A 2-D Table of a Function With Respect to a Partition of its Variables

matter which column is specified by x_1x_2 . In other words, the value of x_1 and x_2 do not impact the value of f . The value of f depends only on the value of x_3 and x_4 (i.e. which row of the 2-D table). Therefore, there exists a function $F : \{0, 1\}^2 \rightarrow \{0, 1\}$ such that $F(x_3, x_4) = f(x_1, x_2, x_3, x_4)$ for all x_1 in X_1 , all x_2 in X_2 , all x_3 in X_3 , and all x_4 in X_4 . In a very reasonable sense then, F is a complete representation of f .

Another way of saying that all the columns of f 's partition matrix are the same is to say that there is only one distinct column in f 's partition matrix. The number of distinct columns is an important part of the basic decomposition condition and is called the *column multiplicity* and denoted ν .

From the above example, we see that if $\nu = 1$ then the variables associated with the columns of the partition matrix can be dropped. That is, if $\nu = 1$ then there exists a function (F) on the row variables only, which is equal to f . We can begin to see the relationship between ν and the decomposability of a function.

Now consider the function g defined in Table 5.6. With respect to the partition of variables $v_1 = \{x_1, x_2\}$ and $v_2 = \{x_3, x_4\}$, g has the partition matrix of Table 5.7.

Notice that there are two distinct columns in g 's partition matrix. That is, $\nu = 2$. In this case it is not possible to drop the v_1 (i.e. x_1 and x_2) variables. Without the v_1 variables there is an ambiguity in the value of the function when $(x_3, x_4) = (0, 1)$ or $(1, 1)$. Even though we cannot drop the v_1 variables, the v_1 variables are really only needed to distinguish between the two distinct columns. Therefore, we can define a function $\phi : X_1 \times X_2 \rightarrow Z$ which selects the appropriate column when given x_1 and x_2 . For example, $z = 0$ indicates the first column and $z = 1$ indicates the second column. There also exists a function G which takes z (to select between the two distinct columns), x_3 and x_4 as input, and represents g , that is, $g(x_1, x_2, x_3, x_4) = G(\phi(x_1, x_2), x_3, x_4)$. See Table 5.8 and Table 5.9 where g is defined using G and ϕ .

From the preceding example, we see that if $\nu = 2$ for a function g with respect to the partition (v_1, v_2) then there exist functions G and ϕ such that $g(v_1, v_2) = G(\phi(v_1), v_2)$.

Summarizing the prior two examples, if $\nu = 1$, then the v_1 variables can be dropped (or reduced to a "unary" variable) and if $\nu = 2$ then the v_1 variables can be reduced to a binary variable. A trend in the relationship between ν and decomposability is beginning to develop.

x_1	x_2	x_3	x_4	$g(x_1, x_2, x_3, x_4)$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 5.6: A Table Representation of Function g

		x_1x_2			
g		00	01	10	11
x_3x_4	00	1	1	1	1
	01	0	1	1	0
	10	0	0	0	0
	11	1	0	0	1

Table 5.7: A Partition Matrix (2-D Table) of g

		x_1x_2			
g		00	01	10	11
$\phi(x_1, x_2)$		0	1	1	0
x_3x_4	00	1	1	1	1
	01	0	1	1	0
	10	0	0	0	0
	11	1	0	0	1

Table 5.8: A Partition Matrix of g With ϕ Defined

	G	ϕ	
		0	1
x_3x_4	00	1	1
	01	0	1
	10	0	0
	11	1	0

Table 5.9: g Defined by G and ϕ

Consider one final example, which we only outline. Suppose $h : X_1 \times X_2 \times X_3 \times X_4 \rightarrow Y$ has $\nu = 3$ for the partition of variables $v_1 = \{x_1, x_2\}$ and $v_2 = \{x_3, x_4\}$. Then, as in the second example, we can define $\eta : X_1 \times X_2 \rightarrow Z$ and $H : Z \times X_3 \times X_4 \rightarrow Y$ such that $h(x_1, x_2, x_3, x_4) = H(\eta(x_1, x_2), x_3, x_4)$, except now we must have three elements in Z (e.g. $Z = \{0, 1, 2\}$) to distinguish the three distinct columns. Clearly, for any function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$, as long as Z has as many as ν elements there will always exist functions $\phi : V_1 \rightarrow Z$ and $F : Z \times V_2 \rightarrow Y$ such that $f(x_1, x_2, \dots, x_n) = F(\phi(v_1), v_2)$. For example, if $\nu = 4$ then $Z = \{0, 1, 2, 3\}$ is sufficient for a decomposition of the form of the prior examples. This is, essentially, the decomposition condition. Once ν is determined for a partition of a function's variables, we know how big Z must be for the function to decompose with respect to that partition. In particular, for any function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$, if $\nu \leq [Z]$ with respect to variable partition v_1, v_2 , then there exist functions $\phi : V_1 \rightarrow Z$ and $F : Z \times V_2 \rightarrow Y$ such that $f(x_1, x_2, \dots, x_n) = F(\phi(v_1), v_2)$.

In the above examples we assumed that Z is of the form $\{1, 2, 3, \dots, [Z]\}$. We could have used a set of vectors for Z , e.g. $Z = \{0, 1\}^2 = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. When Z is a vector set, we can think of ϕ as a single vector valued function, or as a vector of scalar valued functions. For example, if $V_1 = \{0, 1\}^3$, V_2 is $\{0, 1\}$, and $\nu = 4$, then we could define any of the following decompositions: $f(x_1, x_2, x_3, x_4) = F(\phi(x_1, x_2, x_3), x_4) = F'(\phi'(x_1, x_2, x_3), x_4) = F''(\phi''(x_1, x_2, x_3)\phi'''(x_1, x_2, x_3), x_4)$ where $\phi : V_1 \rightarrow \{0, 1, 2, 3\}$, $\phi' : V_1 \rightarrow \{0, 1\}^2$, $\phi'' : V_1 \rightarrow \{0, 1\}$, and $\phi''' : V_1 \rightarrow \{0, 1\}$. Table 5.10 shows examples for the various ϕ 's. Therefore, a basic decomposition does not necessarily have exactly two component functions. In particular, there can be several ϕ 's.

When Z is a vector set, its cardinality is the product of the cardinalities of the sets in the product of sets forming Z . That is, if $Z = X_1 \times X_2 \times X_3 \times \dots \times X_k$ then $[Z] = [X_1][X_2][X_3]\dots[X_k]$. For cost considerations (discussed later) we are interested in keeping $[Z]$ as small as possible. We also want to maximize k since this will provide more opportunities for decomposing F . The ideal situation is for ν to be some power of 2; in this case $Z = \{0, 1\}^k$, where $k = \log(\nu)$, meets both of our objectives. However, if we insist on using a product of some set X then when ν is just slightly larger than a power of $[X]$, $[Z]$ is almost twice as large as really required. Theorem 5.1 provides one possible point in the $[Z]$ and k trade-off.

x_1	x_2	x_3	ϕ	ϕ'	ϕ''	ϕ'''
0	0	0	0	(0,0)	0	0
0	0	1	2	(1,0)	1	0
0	1	0	1	(0,1)	0	1
0	1	1	1	(0,1)	0	1
1	0	0	0	(0,0)	0	0
1	0	1	3	(1,1)	1	1
1	1	0	2	(1,0)	1	0
1	1	1	0	(0,0)	0	0

Table 5.10: Various Forms of Z

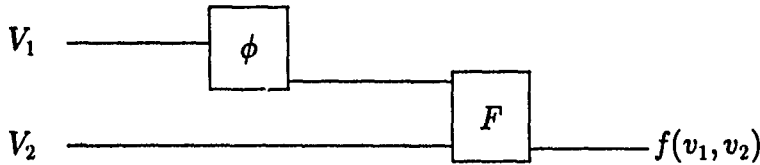


Figure 5.1: Form of a Decomposition

Theorem 5.1 For minimum $[Z]$, (i.e. $[Z] = \nu$), the number of variables in $Z = X_1 \times X_2 \times X_3 \times \dots \times X_k$ is maximized if $[X_1], [X_2], [X_3], \dots, [X_k]$ are the prime factors of ν .

There is one final twist to the decomposition problem that we want to discuss at an intuitive level. In the decomposition condition just discussed, we were considering decompositions of the form of Figure 5.1.

A generalization of this form is to allow some of the variables which are inputs to ϕ to also be inputs to F ; that is, to allow decompositions of the form of Figure 5.2.

In this more general case the partition matrix is three-dimensional. V_1 defines columns as before. V_3 defines rows as before. V_2 defines the third dimension. That is, for every value of V_2 , there is a traditional two-dimensional partition matrix. The information of V_2 is available to F ; differences in the values of the function across

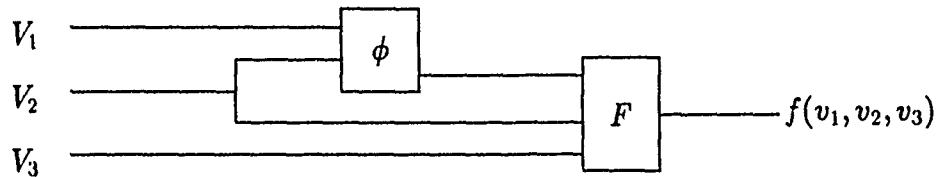


Figure 5.2: Form of a More General Decomposition

			x_1x_2			
			00	01	10	11
$x_3 = 0$	x_4	x_5	0	0	1	1
	0	1	0	0	1	1
	1	0	0	0	1	1
	1	1	0	0	1	1
$x_3 = 1$	0	0	0	1	0	1
	0	1	0	1	0	1
	1	0	0	1	0	1
	1	1	0	1	0	1

Table 5.11: Partition Matrices

this third dimension do not have to be reflected in ϕ , that is, they can be accounted for in F . Therefore, in defining ϕ , it is only necessary that each individual layer (for each value of V_2) be decomposable. Let ν_{v_2} represent the column multiplicity of the 2-D matrix for v_2 . The decomposition condition therefore is $\nu_{v_2} \leq [Z]$ for all v_2 in V_2 . If we redefine ν to be the maximum of all ν_{v_2} for any v_2 in V_2 then the condition can be given as $\nu \leq [Z]$. The V_2 variables are "shared" between ϕ and F ; thus we sometimes call this generalization a "shared variable" decomposition.

As an example of shared variable decomposition consider the function

$$f: \{0, 1\}^5 \rightarrow \{0, 1\}$$

defined in the partition matrices of Table 5.11. The partition corresponding to Table 5.11 is $V_1 = \{x_1, x_2\}$, $V_2 = \{x_3\}$, and $V_3 = \{x_4, x_5\}$.

From the partition matrices we see that $\nu_{x_3=0} = 2$ and $\nu_{x_3=1} = 2$, thus $\nu = 2$. A decomposition exists with respect to this partition when $Z = \{0, 1\}$, e.g. Figure 5.3.

5.2.3 The Formal Basic Decomposition Condition

Introduction

When discussing decompositions of a function it is natural to classify different kinds of decompositions according to their properties. An extensive taxonomy of decompositions is developed in [14] (e.g. there are simple, multiple, iterative, disjunctive, and proper types of decompositions as well as their complements and many of their combinations). Without detracting in any way from the importance of these classes, we propose a new class called the basic decompositions. This class is defined to correspond exactly to the class of decompositions which can be tested for with the decomposition condition to be developed in this report.

We now formally define a basic decomposition. Let n be a finite integer and let X_1, X_2, \dots, X_n , and Y be finite sets, each of cardinality two or more. We are

x_1x_2	$\phi_{x_3=0}$	$\phi_{x_3=1}$
00	0	0
01	0	1
10	1	0
11	1	1

$x_1x_2x_3$	ϕ
000	0
001	0
010	0
011	1
100	1
101	0
110	1
111	1

Figure 5.3: Example Decomposition

interested in any partial function f of the form $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$. Let x_1, x_2, \dots, x_n represent variables of the sets X_1, X_2, \dots, X_n , respectively. That is, x_i is some unspecified element of X_i . Let V_1, V_2 and V_3 be products of finite sets, say $V_1 = V_{11} \times V_{12} \times \dots \times V_{1n'}$, $V_2 = V_{21} \times V_{22} \times \dots \times V_{2n''}$, and $V_3 = V_{31} \times V_{32} \times \dots \times V_{3n'''}$, where there exists a bijection $b : \{1, 2, \dots, n\} \rightarrow \{11, 12, \dots, 1n', 21, 22, \dots, 2n'', 31, 32, \dots, 3n'''\}$ such that $X_i = V_{b(i)}$ for $i = 1, 2, \dots, n$. v_1, v_2, v_3 are variables associated with V_1, V_2 , and V_3 , respectively. The variable v_i represents the variables $v_{i1}, v_{i2}, \dots, v_{in'}$. V_1, V_2 and V_3 are a partition of the domain of f . By a "partition" we mean that except for possibly the order of the sets in the products, $X_1 \times X_2 \times \dots \times X_n = V_1 \times V_2 \times V_3$. Consider $f' : V_1 \times V_2 \times V_3 \rightarrow Y$ such that $f(x_1, x_2, \dots, x_n) = f'(v_1, v_2, v_3)$ when v_i 's and the x_j 's are related by b . Rather than distinguishing f and f' , we simply use f for either function; which function is made clear by the context. That is, we say $f(x_1, x_2, \dots, x_n)$ and $f(v_1, v_2, v_3)$, when v_1, v_2, v_3 is a partitioning of the variables x_1, x_2, \dots, x_n . We similarly use $f(v_{11}, v_{12}, \dots, v_{1n'}, v_{21}, v_{22}, \dots, v_{2n''}, v_{31}, v_{32}, \dots, v_{3n'''})$ with the obvious meaning.

A basic decomposition of a function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ with respect to the partition V_1, V_2 and V_3 is two functions $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ such that $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ for all $v_1 \in V_1, v_2 \in V_2$, and $v_3 \in V_3$ when $f(v_1, v_2, v_3)$ is defined.

When f is not a total function, we only require that $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ when f is defined. That is, $F(\phi(v_1, v_2), v_2, v_3)$ may be defined arbitrarily or undefined whenever $f(v_1, v_2, v_3)$ is undefined. Our justification for this comes from two sources. First, when partial functions arise in practice, the elements with undefined images either cannot occur or when they do occur we do not care what the function outputs for that input. Therefore, allowing a decomposition to produce a value when the original function was undefined is often acceptable in practice. Secondly, in those cases where we do want the decomposition to be undefined whenever the original function was undefined, we can modify the semantics of the problem slightly and make it a special case of what we allow. To modify the semantics of the problem, we define a total func-

tion $f' : X_1 \times X_2 \times \dots \times X_n \rightarrow Y \cup \{u\}$ such that $f'(x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n)$ when f is defined and $f'(x_1, x_2, \dots, x_n) = u$ when f is undefined. The decomposition properties of f' with respect to our criteria are the same as the decomposition properties of f when undefined values must be preserved.

Some of the more familiar types of decompositions are special cases of basic decompositions. For example, when $X_i = \{0, 1\}$ for $i = 1, 2, \dots, n$ and $Y = \{0, 1\}^m$, we have the classical Boolean functions. When V_2 is empty and $Z = \{0, 1\}$ we have a traditional "nondisjunctive" decomposition. If $Z = \{0, 1\}^k$, for some integer k , we might think of ϕ as a vector function (or k distinct functions on the same domain). In this case we have an "improper" decomposition. A basic decomposition can be any of the Curtis types of decomposition. Multiple-valued logic functions are obviously covered by this form as well.

The Basic Decomposition Condition Theorem

Before stating the basic decomposition condition, we need to develop a formal definition of the column multiplicity (ν) of a partition matrix given a function f and a partition of its variables V_1, V_2, V_3 . We assume that V_1 and V_3 are both non-empty. When either is empty, F or ϕ is exactly f ; therefore, no real "decomposition" is involved. Since V_3 is a finite set of $[V_3]$ elements, we can define a bijection $b : \{1, 2, \dots, [V_3]\} \rightarrow V_3$. A "column" ($C_{v_1 v_2}$) for some fixed v_1 and v_2 is defined as the sequence: $C_{v_1 v_2} = (f(v_1, v_2, b(1)), f(v_1, v_2, b(2)), f(v_1, v_2, b(3)), \dots, f(v_1, v_2, b([V_3])))$. Therefore, $C_{v_1 v_2}$ is a vector from $Y^{[V_3]}$. Columns form a "set of columns" (S_{v_2}) for a fixed $v_2 : S_{v_2} = \{C_{v_1 v_2} | v_1 \in V_1\}$. When f is total, ν is the maximum element of $\{|S_{v_2}| | v_2 \in V_2\}$; however, when f is not total, we need an extra step.

Call two columns compatible if the only coordinates in which they differ are those where either is undefined. Consider $\hat{S}_{v_2} = \{C_{v_1 v_2} | f(v_1, v_2, v_3) \text{ is defined for each } v_1, v_3\}$. If \hat{S}_{v_2} is empty, we can go to later steps. If not, define relation \hat{R}_{v_2} on \hat{S}_{v_2} using prefix notation by $\hat{R}_{v_2}(C_{v_1 v_2}, C_{v_1' v_2})$ if $f(v_1, v_2, v_3) = f(v_1', v_2, v_3)$ for all $v_3 \in V_3$. This is an equivalence relation on S_{v_2} . Enumerate the resulting equivalence classes $E_1, E_2, \dots, E_{i_{v_2}}$ calling representatives $e_1, e_2, \dots, e_{i_{v_2}}$. Also enumerate the elements of the set $S_{v_2} \setminus \hat{S}_{v_2}, C_1, C_2, \dots, C_{j_{v_2}}$. Choose the first class such that C_1 is compatible with the representative of that class and adjoin C_1 to that class. If no such class is found, C_1 will belong to its own class $E_{i_{v_2}+1}$. If C_1 creates a new class, it is the representative - called $e_{i_{v_2}+1}^1$. Otherwise, define the new representative of the class that C_1 is in, to be e_k^1 where $(e_k^1)_m$ is the value of $(C_1)_m$ or $(e_k)_m$ if either or both are defined (if both, they must be equal), and $(e_k^1)_m$ is undefined if both $(C_1)_m$ and $(e_k)_m$ are undefined. Here, $m = 1$ to $[V_3]$ and m stands for the "coordinate" of the column vectors. Representatives of other classes remain the same but are denoted individually as e_i^1 . Continue in this manner with the other elements of $S_{v_2} \setminus \hat{S}_{v_2}$. That is, if a column C_a is not compatible with any of the existing representatives, it will create a new class whose number is one more than the number of the last class: $i_{v_2} + 1$. It will be the representative, denoted $e_{i_{v_2}+1}^a$. If a column C_a is compatible with an existing representative, we choose the first occurrence of this, adjoin C_a to

that class and proceed as with C_1 except in the above steps (when we dealt with C_1) replace e_k with $e_k^{(a-1)}$, e_k^1 with e_k^a , and e_l^1 with e_l^a . Finally, given v_2 , we have the set of classes $E_{v_2} = \{E_{i_{v_2}} | 1 \leq i_{v_2} \leq \nu_{v_2}\}$ which partitions the set of columns. The classes' representatives are $(e^{j_{v_2}})_1, (e^{j_{v_2}})_2, \dots, (e^{j_{v_2}})_{\nu_{v_2}}$. Call the equivalence relation determined by this partition of S_{v_2}, R_{v_2} . Now we define $\nu_{v_2} = [E_{v_2}]$. When V_2 is empty, there is only one E_{v_2} since v_2 cannot occur in the expression. Finally, we define ν as the maximum over all v_2 in V_2 of ν_{v_2} . This definition relies only on elementary set theory for background.

Theorem 5.2 (The Basic Decomposition Condition) ¹ For finite integer n , and finite sets $X_1 \times X_2 \times \dots \times X_n, Y$, let f be a partial function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$. Let V_1, V_2 , and V_3 be a partition of the domain of f . There exist functions $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ such that, whenever $f(v_1, v_2, v_3)$ is defined, $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ if and only if $\nu \leq [Z]$.

Proof:

First we prove that $\nu \leq [Z]$ implies that there exist functions $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ such that, whenever $f(v_1, v_2, v_3)$ is defined, $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$.

Step 1. Define $\phi : V_1 \times V_2 \rightarrow Z$ by $\phi(v_1, v_2) = i$ if $C_{v_1 v_2} \in E_{i_{v_2}}$.

Step 2. Define $F : Z \times V_2 \times V_3 \rightarrow Y$ as follows:

i. if $z = i$ for some i , then $F(i, v_2, v_3) = (e_{i_{v_2}}^{j_{v_2}})_{b(v_3)}$ where b is the bijection from V_3 into $\{1, 2, \dots, [V_3]\}$,

ii. otherwise, $F(z, v_2, v_3)$ is undefined. Once in place, defined coordinates of the representatives of the equivalence classes do not change, so when defined $f(v_1, v_2, v_3) = (e_{\phi(v_1, v_2)}^{j_{v_2}})_{b(v_3)} = F(\phi(v_1, v_2), v_2, v_3)$.

Now we prove that the existence of functions $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ such that, whenever $f(v_1, v_2, v_3)$ is defined, $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ implies that $\nu \leq [Z]$. First observe that $\nu \leq [Z]$ is logically equivalent to $\nu_{v_2} \leq [Z]$ for all $v_2 \in V_2$. Assume to the contrary that there exists $v_2 \in V_2$ such that $\nu_{v_2} > [Z]$.

1) We can assume that equivalent columns correspond to ordered pairs which have the same inverse image under ϕ . When $C_{v_1 v_2}$ and $C_{v_1' v_2}$ are R_{v_2} -related, then when defined $f(v_1, v_2, v_3) = f(v_1', v_2, v_3)$ for each v_3 . Thus, $F(\phi(v_1, v_2), v_2, v_3) = F(\phi(v_1', v_2), v_2, v_3)$. No harm is done to the relationship between ϕ and F and the range of ϕ is no larger than before. Hence, we assume that the inverse image of an element of Z contains ordered pairs which correspond to an equivalence class (under R_{v_2}) of columns.

2) If $\nu_{v_2} > [Z]$, then there must be two non-equivalent columns whose corresponding ordered pairs have the same image under ϕ . These columns cannot come from S_{v_2} since R_{v_2} is the equality relation there because if two columns from S_{v_2} are not equivalent, there must exist a v_3 such that $f(v_1, v_2, v_3) \neq f(v_1', v_2, v_3)$. Hence,

¹Mike Breen contributed substantially to the development of this proof. See [4] for the original development of a decomposition condition.

$f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3) = F(\phi(v_1', v_2), v_2, v_3) = f(v_1', v_2, v_3) \neq f(v_1, v_2, v_3)$, a contradiction.

3) For $S_{v_2} \setminus \hat{S}_{v_2}$, use the ordering developed in defining R_{v_2} . Choose the first element of $S_{v_2} \setminus \hat{S}_{v_2}, C_j$, such that the number of equivalence classes in $\hat{S}_{v_2} \cup \{C_1, \dots, C_j\}$ is larger than $[\phi(\hat{S}_{v_2} \cup \{C_1, \dots, C_j\})]$. If no such element exists, $\nu_{v_2} < [Z]$. Because of how C_j was chosen, C_j must create its own class. That is, C_j is not compatible with any representative of the equivalence classes at that time. We can write $C_{v_1 v_2}$ for C_j and see that there exists $C_{v_1' v_2} \in \hat{S}_{v_2} \cup \{C_1, \dots, C_j\}$ such that $\phi(v_1, v_2) = \phi(v_1', v_2)$. Call the representative of the class of $C_{v_1' v_2}, e^a$. We know that $C_{v_1 v_2}$ is not compatible with e^a . Therefore, there is a $C_{v_1'' v_2}$ currently in the class with $C_{v_1' v_2}$ such that for some $v_3, f(v_1'', v_2, v_3)$ and $f(v_1, v_2, v_3)$ are defined and unequal. This last statement follows from the fact that a column is compatible with a representative of a class if and only if it is compatible with each element in the class. Now we have $R_{v_2}(C_{v_1'' v_2}, C_{v_1' v_2})$ which implies that $\phi(v_1'', v_2) = \phi(v_1', v_2) = \phi(v_1, v_2)$ and that $F(\phi(v_1'', v_2), v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ for each v_3 . Yet for some $v_3, f(v_1'', v_2, v_3) = f(v_1, v_2, v_3)$ and both are defined. As before, this contradicts the assumption about ϕ and F . Hence, no such non-equivalent columns exist.

□

5.2.4 Non-Trivial Basic Decompositions

For a mapping of a given form $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ and for a given partition V , if $[Z]$ is sufficiently large then every possible function of that form will decompose with respect to that partition. We call decompositions of this type trivial. Decompositions which are not trivial are called non-trivial. Since non-trivial decompositions do not always exist, they are in some sense special. This section develops the condition for the existence of non-trivial basic decompositions.

First we establish the least upper bound on ν . Define ν_{\max} to be the smaller of $[V_1]$ or $[Y]^{[1]}$.

Theorem 5.3 *If $V = \{V_1, V_2, V_3\}$ is a partition of the variables of the function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ then ν of f with respect to that partition is less than or equal to ν_{\max} . Further, there is no other bound less than ν_{\max} .*

Proof:

First we show that $\nu \leq [V_1]$. $[V_1]$ is the total number of columns in the partition matrix, therefore the number of distinct columns cannot exceed this number. More rigorously, $[\{R_{v_1 v_2} | v_1 \in V_1\}] \leq [V_1]$ for all v_2 , since there must be an element in the right hand set for every element in the left hand set. By definition of E_{v_2} then, $[E_{v_2}] \leq [V_1]$ for all v_2 and by definition of ν_{v_2} , $\nu_{v_2} \leq [V_1]$ for all V_2 . Finally, by definition of ν , $\nu \leq [V_1]$.

Now we show that $\nu \leq [Y]^{[1]}$. $[V_3]$ is the number of rows in the partition matrix, and the partition matrix contains elements from Y . Therefore, each column is a string

of characters from Y of length $[V_3]$ and there can only be $[Y]^{[V_3]}$ such strings. More rigorously, recall that $C_{v_1 v_2} = (f(v_1, v_2, b(1)), f(v_1, v_2, b(2)), \dots, f(v_1, v_2, b([V_3])))$. Let A be the set of all possible $C_{v_1 v_2}$'s. $C_{v_1 v_2}$ is a string on Y of length $[V_3]$, therefore, $[A] = [Y]^{[V_3]}$. By definition of S_{v_2} , S_{v_2} is a subset of A , thus $[S_{v_2}] \leq [A] = [Y]^{[V_3]}$. E_{v_2} is a partition of S_{v_2} and does not contain the empty set so $[E_{v_2}] \leq [S_{v_2}]$. By definition of ν_{v_2} , $\nu_{v_2} = [E_{v_2}] \leq [S_{v_2}] \leq [Y]^{[V_3]}$. This is true for arbitrary ν_{v_2} , therefore $\nu \leq [Y]^{[V_3]}$.

We have shown that $\nu \leq [V_1]$ and that $\nu \leq [Y]^{[V_3]}$. therefore ν is less than or equal to the smaller of these two limits. That is, $\nu \leq \nu_{\max}$.

There is no bound less than ν_{\max} since we can always construct a function of the given form with respect to the given partition such that the partition matrix contains distinct columns up to the ν_{\max} limit.

□

We can now develop the sufficient condition on $[Z]$ such that decompositions always exist.

Theorem 5.4 $F : Z \times V_2 \times V_3 \rightarrow Y$ and $\phi : V_1 \times V_2 \rightarrow Z$ is a trivial decomposition of $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ with respect to partition V if and only if $[Z] \geq \nu_{\max}$.

Proof:

First we prove that $[Z] \geq \nu_{\max}$ implies that the decomposition is trivial. Assume that $[Z] \geq \nu_{\max}$. By Theorem 5.3, $\nu \leq \nu_{\max}$; with the assumption, this becomes $[Z] \geq \nu_{\max} \geq \nu$. Finally, from the Basic Decomposition Condition, $\nu \leq [Z]$, we see that decomposition is always possible.

To prove the implication the other way: Assume that the decomposition is trivial; we want to show that $[Z] \geq \nu_{\max}$. Suppose to the contrary that $[Z] < \nu_{\max}$. This $[Z]$ would be an upper bound on ν , since this decomposition is possible for all functions. But this bound is less than ν_{\max} , which violates the Theorem 5.3. Therefore, the supposition is false and the theorem follows.

□

The relationship between $[V_1]$ and ν is shown graphically in Figure 5.4. We use the fact that $[f] = [V_1][V_2][V_3]$ to plot ν as a function of $[V_1]$ with $[f]$ and $[V_2]$ as the only parameters.

The following theorem is the non-trivial basic decomposition condition.

Theorem 5.5 For finite integer n , and finite sets $X_1 \times X_2 \times \dots \times X_n, Y$, let f be a partial function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$. Let V_1, V_2 , and V_3 be a partition of the domain of f . There exists a non-trivial decomposition of f consisting of the functions $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ such that, whenever $f(v_1, v_2, v_3)$ is defined, $f(v_1, v_2, v_3) = F(\phi(v_1, v_2), v_2, v_3)$ if and only if $\nu < \nu_{\max}$.

Proof:

From the Basic Decomposition Condition we have $\nu \leq [Z]$ and from the Trivial

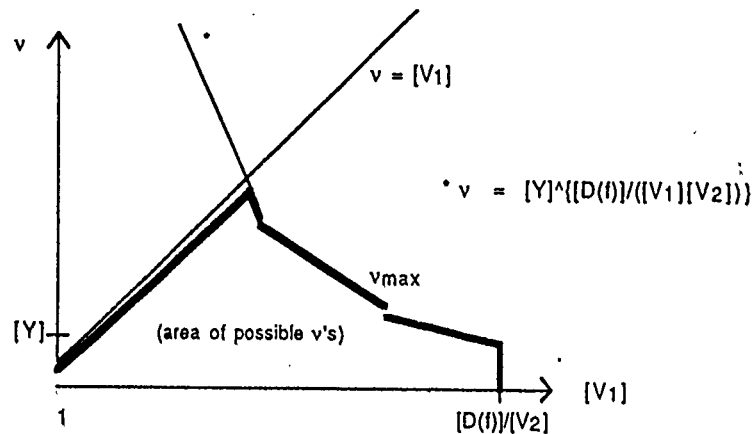


Figure 5.4: Relationship Between ν and $[V_1]$, where $D(f) = [f]$

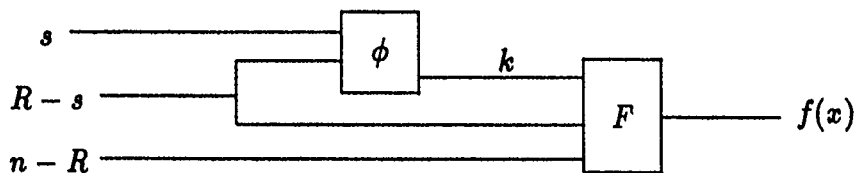


Figure 5.5: The Basic Decomposition

Decomposition Theorem, we have that a decomposition is non-trivial if and only if $[Z] < \nu_{\max}$.

□

We now develop one special case of the above theorem in some detail. Assume that $f : X^n \rightarrow X$ and we require that $Z = X^k$. The Basic Decomposition Condition becomes $\nu \leq [X]^k$. Define parameter s and R such that $[V_1] = [X]^s$ and $[V_2] = [X]^{(R-s)}$. The parameters n, s, R and k are the number of variables in their respective groups (see Figure 5.5).

In this special case ν_{\max} becomes the smaller of $[X]^s$ and $[X]^{[X]^{(n-R)}}$ or equivalently, $k_{\max} = \min(s, [X]^{(n-R)})$. Therefore, the Basic Decomposition Condition is always satisfied if $s \leq k$ or $[X]^{(n-R)} \leq k$. The special case of this where $k = 1$ and $s = 0$ or $s = 1$ is the "trivial decomposition" of Curtis'62.

x_1	x_2	x_3	f	g
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

Table 5.12: Functions f and g

5.2.5 Negative Basic Decompositions

In decomposing a function we are often interested in minimizing the cost of a realization of the function based on that decomposition. We pursue two particular cost functions.

We use r_f to denote the representation of f . $DFC(r_f)$ is the decomposed function cardinality of that representation. $DFC(f)$ is the $DFC(r_f)$ when r_f is the optimal representation of f . Similarly, $L(r_f)$ is the program length of r_f and $L(f)$ is the optimal program length of f .

Our first cost function $L(r_f) = 1 + 3n + 2(n+1)[A] + n[P] + DFC(f) = 1 + 5n + 2(n+1)([V_1] + 2[V_2] + [V_3] + 1) + [\phi] \log[Z] + [F] \log[Y]$ reflects interconnection complexity and DFC. It is possible to completely define any representation r_f with $L(r_f)$ bits.

The second cost function $DFC(r_f) = [\phi] \log[Z] + [F] \log[Y]$ is a reflection of size complexity only. However, it is demonstrated in Chapter 4 that size complexity, as defined by DFC, is the principal component of overall complexity and that DFC is fundamentally tied to time complexity, circuit complexity and program length.

A decomposition is "negative" if its cost is less than the cost of the un-decomposed function. L -Negative implies non-trivial.

Theorem 5.6 *Being non-trivial is a necessary but not a sufficient condition for a basic decomposition to be negative.*

Proof:

Non-triviality is a necessary condition since if a negative decomposition were trivial then every function would have a negative decomposition and the average minimum cost would violate the Average-Minimum Program Length Lower Bound of Appendix A. That non-triviality is not sufficient can be demonstrated with an example. Consider $f : \{0,1\}^3 \rightarrow \{0,1\}$ defined in Table 5.12

The partition $V_1 = \{x_1, x_2\}$, $V_2 = \emptyset$, and $V_3 = \{x_3\}$ has $\nu = 2$ for f and $\nu = 4$ for g . f has a decomposition of the form $\phi : \{0,1\}^2 \rightarrow \{0,1\}$ and $F : \{0,1\}^2 \rightarrow \{0,1\}$, i.e.

$Z = \{0, 1\}$. This decomposition is non-trivial since there exists a function (namely g) that does not have this decomposition with respect to this partition. This decomposition is also not negative. That is, $L(r_f) \geq DFC(r_f) = 2^2 + 2^2 = 8$ which is not less than $2^3 = 8$.

□

Explicit statements of the various negative decomposition conditions are of interest since they are used in decomposition algorithms. The following is called the *L-Negative Basic Decomposition Condition*.

Theorem 5.7 *A Basic Decomposition $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ of a function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ with respect to partition V is negative with respect to L if and only if $1 + 7n + 2(n + 1)([V_1] + 2[V_2] + [V_3]) + [V_1][V_2]\log[Z] + [Z][V_2][V_3]\log[Y] < [V_1][V_2][V_3]\log[Y]$.*

Proof:

By definition of a negative decomposition: $L(r_f) < [X]\log[Y] + 1$. The theorem follows by substitution and simplification.

□

The next theorem is the *DFC-Negative Basic Decomposition Condition*.

Theorem 5.8 *A Basic Decomposition $\phi : V_1 \times V_2 \rightarrow Z$ and $F : Z \times V_2 \times V_3 \rightarrow Y$ of a function $f : X_1 \times X_2 \times \dots \times X_n \rightarrow Y$ with respect to partition V is negative with respect to DFC if and only if $[V_1]\log[Z] + [Z][V_3]\log[Y] < [V_1][V_3]\log[Y]$.*

Proof:

By definition of a negative decomposition: $DFC(r_f) < [X]\log[Y]$. The theorem follows by substitution and simplification.

□

The largest $[Z]$ that satisfies the above inequalities is the largest $[Z]$ that will yield a negative decomposition in the applicable situation. Since $[Z]$ must be greater than or equal to the column multiplicity ν , the above inequalities give the maximum ν that will result in a negative decomposition. Therefore, when we are exclusively interested in negative decompositions, we are only interested in ν 's which satisfy the above inequalities. That is, in the *DFC* case, $[V_1]\log \nu + \nu[V_3]\log[Y] < [V_1][V_3]\log[Y]$. If ν_{\max} is the largest integer that satisfies the negative decomposition condition then we are only interested in ν 's which are less than or equal to ν_{\max} . Therefore, when we are counting columns in a partition matrix, we can stop counting as soon as ν reaches ν_{\max} . With respect to our special case of the previous section (i.e. $f : X^n \rightarrow X$, $Z = X^k$, $V_1 = X^s$, $V_2 = X^{(R-s)}$, and $V_3 = X^{(n-R)}$) the *DFC-Negative* decomposition condition becomes $k[X]^s + [X]^{(k+n-R)} < [X]^R$. That is, from the theorem above:

$$[V_1]\log[Z] + [Z][V_3]\log[Y] < [V_1][V_3]\log[Y],$$

$$\begin{aligned}
[X]^s \log([X]^k) + [X]^k [X]^{(n-R)} \log[X] &< [X]^s [X]^{(n-R)} \log[X], \\
k[X]^s \log[X] + [X]^{(k+n-R)} \log[X] &< [X]^{(n-R+s)} \log[X], \\
k[X]^s + [X]^{(k+n-R)} &< [X]^{(n-R+s)}.
\end{aligned}$$

With the further specialization that there be no shared variables (i.e. $R = s$), the condition becomes $k[X]^R + [X]^{(k+n-R)} < [X]^{(n)}$ or $k[X]^{(R-n)} + [X]^{(k-R)} < 1$.

If we define k_{\max} as the largest integer satisfying the applicable condition then we know that k must be less than or equal to k_{\max} for negative decompositions. It follows then that the maximum ν of interest is $\nu_{\max} = [X]^{k_{\max}}$.

The important point is that, given n and V , we can directly determine the maximum ν for a negative decomposition. This test can be useful since we may want to discontinue counting columns in evaluating column multiplicity after we are assured that no negative decomposition is possible.

We have one final result concerning basic decompositions. Let N be the number of ways that a set X can be partitioned into three sets V_1, V_2, V_3 .² The number of combinations of i elements from the set X , where X has n elements, is $n!/(n-i)! = C(n, i)$. This takes care of the first set V_1 . Now select j elements from $n-i$ elements: $(n-i)!/j!(n-i-j)! = C(n-i, j)$ where j elements go into V_2 and $(n-i-j)$ elements go into V_3 . Thus the number of combinations for one partition is $C(n, i)C(n-i, j)$.

For each i there are different combinations of $n-i$ elements, so for all partitions, the number of different combinations is:

$$\begin{aligned}
N &= \sum_{i=0}^n \sum_{j=0}^{n-i} C(n, i) C(n-i, j) \\
&= \sum_{i=0}^n C(n, i) \sum_{j=0}^{n-i} C(n-i, j) \\
&= \sum_{i=0}^n C(n, i) 1^i \sum_{j=0}^{n-i} C(n-i, j) 1^{(n-i-j)} 1^{(j)} = \sum_{i=0}^n C(n, i) 1^i (1+1)^{(n-i)}.
\end{aligned}$$

Recalling the Binomial Theorem [41, p.27]:

$$(a+b)^n = \sum_{i=0}^n C(n, i) a^{(n-i)} b^i,$$

$$N = \sum_{i=0}^n C(n, i) 1^i 2^{(n-i)} = (1+2)^n = 3^n.$$

The number of nontrivial partitions (N') can be derived by the same method. We must make sure that V_1 has no less than two elements and no more than $n-2$ elements, so i will go from two to $n-2$. Also V_2 can have zero elements, but no more

²This result was developed by Tina Normand.

than $n - i - 2$ elements, (to make sure V_3 has at least 2 elements), so j will go from zero to $n - i - 2$. The result is:

$$\begin{aligned}
 N' &= \sum_{i=2}^{n-2} \sum_{j=0}^{n-i-2} C(n, i) C(n - i, j) \\
 &= \sum_{i=2}^{n-2} C(n, i) \sum_{j=0}^{n-i-2} C(n - i, j) 1^{(n-i)} 1^{(n-i-j)} \\
 &= \sum_{i=2}^{n-2} C(n, i) (2^i - C(n - i, n - i - 1) - C(n - i, n - i)) \\
 &= \sum_{i=2}^{n-2} C(n, i) (2^i - (n - i) - 1) \\
 &= \sum_{i=2}^{n-2} C(n, i) 2^i - (n + 1) \sum_{i=2}^{n-2} C(n, i) + \sum_{i=2}^{n-2} i C(n, i).
 \end{aligned}$$

This can be further reduced since $\sum_{i=0}^n i C(n, i) = n 2^{n-1}$ [65, p71].

5.3 The Ada Function Decomposition Programs

The PT 1 function decomposition algorithm takes a binary partial function of the form $f : \{0, 1\}^n \rightarrow \{0, 1\}$ and attempts to decompose the function into components ϕ and F such that $f(x, y, z) = F(\phi(x, y), y, z)$ where x , y , and z are vectors. We are especially interested in decompositions where the size of the decomposition (i.e. the size of F plus the size of ϕ) is less than the size of the original function. The algorithm accomplishes this decomposition by searching through all possible partitions of input variables and testing to see if the function decomposes for that partition. The test for decomposition and other theoretical aspects of function decomposition are developed in the preceding section. If the function does not decompose with respect to a given partition then another partition is tried. If the function does decompose then an attempt is made to decompose each of the component functions. As the algorithm searches through possible decompositions, any decomposition which has lower cost than any previous decomposition is recorded. When the search is completed, we have the lowest cost decomposition (lowest cost of those considered) of the input function.

Several versions of this function decomposition algorithm were implemented in Ada during the PT 1 project. This section describes these programs. A User's Guide for the AFD program is in Appendix B. The PT 1 function decomposition software was written in Ada on the VMS System running on a Vax 11/780. This Vax is part of the Fire Control Simulation (FICSIM) Facility of WL/AART located in Building 22 at Wright Patterson AFB, Ohio.

DECOMP_RECORD

- A_TABLE ----> TABLE_RECORD
 - VARIABLES ----> LABELS array of
 - NATURALS
 - A_FUNCTION ----> TABLE array of
 - ENTRIES:
(T,F,TBD)
- A_PARTITION ----> PARTITION_RECORD
 - COLUMN_VARIABLES ----> LABELS.
 - ROW_VARIABLES ----> LABELS.
 - NEW_VARIABLES ----> LABELS.
 - A_PARTITION_MATRIX ----> PARTITION_MATRIX array of
 - COLUMNS.
 - FUNCTIONS ----> TABLE_RECORD.
 - UNIQUE_COLUMNS ----> TABLE.
 - MAX_NU: INDEX: INTEGER
 - NU: INDEX.
- LEFT_DECOMP ----> DECOMP_RECORD.
- RIGHT_DECOMP ----> DECOMP_RECORD.
- CHAIN_DECOMP ----> DECOMP_RECORD.
- DECOMP_COST: NATURAL

Figure 5.6: DECOMP_RECORD Data Structure

5.3.1 Program Functional Description

DECOMP_RECORD is the principal data structure for the function decomposition algorithm. The structure of a DECOMP_RECORD is shown in Figure 5.6. Arrows in Figure 5.6 indicate a pointer type and the type being pointed to. Record and array components are preceded by a dash and are indented under the appropriate type. Names followed by a period have additional structure but that structure is shown elsewhere in the figure.

After inputting a function, the algorithm runs a FIND_LOWEST_COST routine and outputs the results. Figure 5.7 is a flow chart for the FIND_LOWEST_COST routine.

Figure 5.8 is a psuedo-code representation of FIND_LOWEST_COST and its principal component DECOMPOSE_CURRENT.

Figure 5.9 shows examples of the main data objects at various stages.

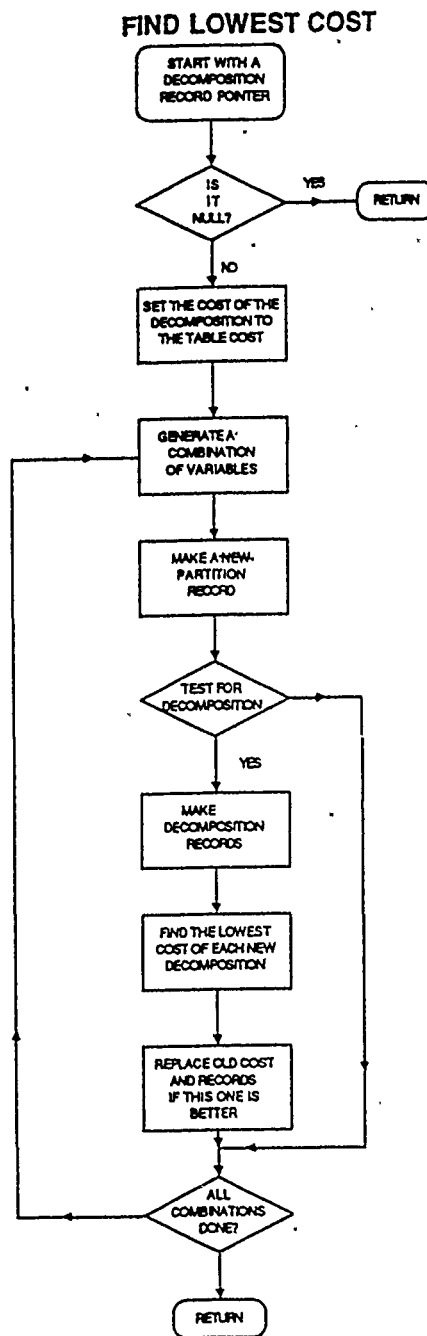


Figure 5.7: FIND_LOWEST_COST Flow Chart

Find Lowest Cost Decomposition Algorithm

```
If the CURRENT Decomposition Record Pointer is null
then return
else
  Begin
    Set BEST=CURRENT
    Repeat
      Generate next combination of variables
      if CURRENT decomposes then DECOMPOSE CURRENT
      FIND LOWEST COST of left, right, and chain functions
      Calculate cost of CURRENT
      if cost of CURRENT < cost of BEST then BEST:=CURRENT
    Until all combinations of variables have been checked
  End
```

Figure 5.8: FIND-LOWEST-COST Psuedo-Code

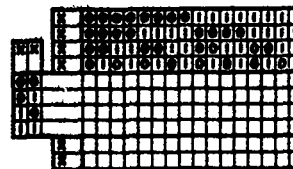
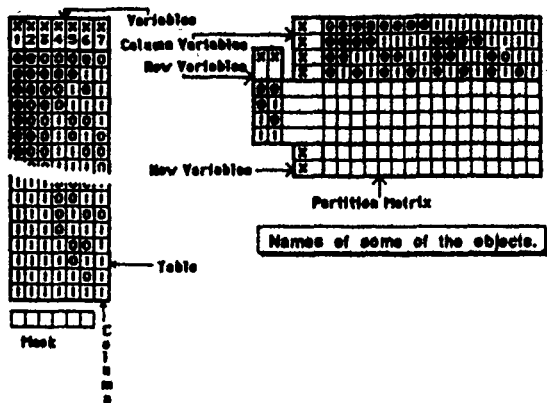
5.3.2 Program Software Description

The main program for the AFD algorithm is the procedure PBML-DRIVER. Procedure PBML-DRIVER uses five packages of software. Package PBML-TYPES defines the data structures and has no body. Package PBML-FREE has 14 procedures for deallocating pointers. Various utility subprograms are in package PBML-UTIL, which contains four procedures and ten functions, and PBML-PACKAGE, which contains 11 procedures and three functions. The input and output routines are in PBML-IO. PBML-IO contains 11 procedures. The AFD software is contained in nine files. There are approximately 1,500 lines of code.

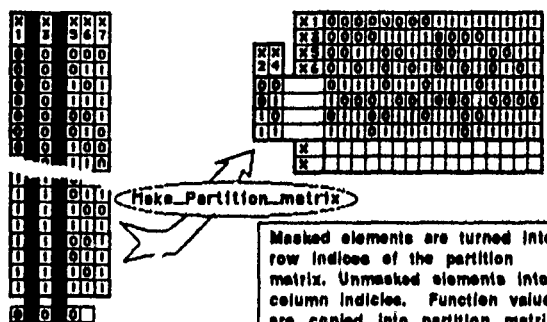
Figure 5.10 represents the compilation dependencies of the AFD software. The arrow means "is dependent upon." For example, PBML-IO should be compiled after PBML-UTIL. After all files have been compiled, PBML-DRIVER should be ACS LINKED and RUN.

5.3.3 Versions of the AFD Algorithm

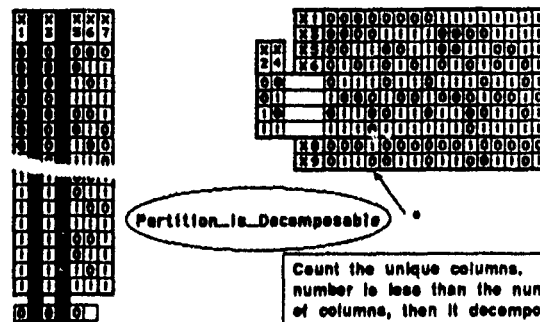
Ten version of the AFD program were implemented. We developed all these versions of the AFD algorithm in hopes of finding two algorithms. We had hoped to find a non-exhaustive optimal algorithm. That is, an algorithm that always finds the lowest possible cost and does so without considering all possible decompositions. We are doubtful that we succeeded in this. We had also hoped to find an algorithm on the knee-of-the-curve of run-time versus decomposition cost. We found that even our fastest versions were able to almost fully decompose most functions.



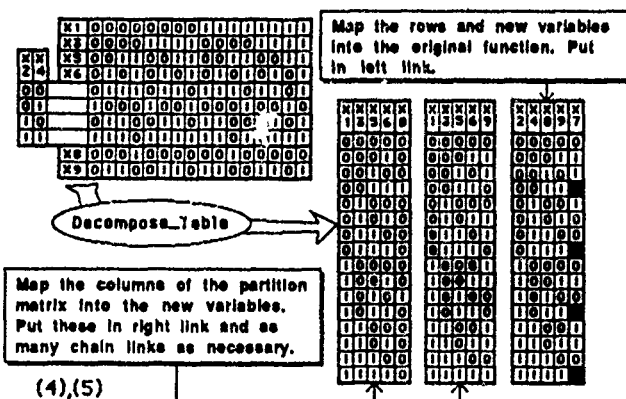
The table values are completely defined by labels and indices. They can be found by operating on the labels and indices rather than specifically using binary columns around. Thus, the 0's and 1's shown here are never actually stored.



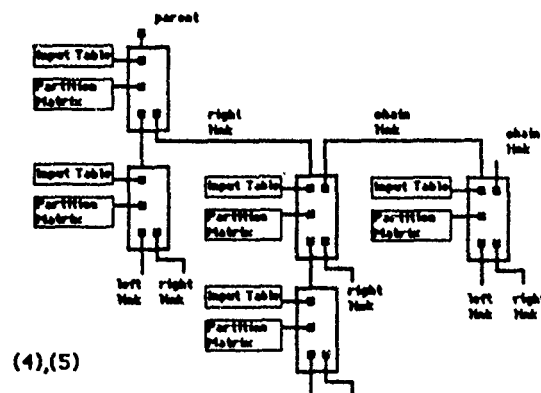
(1)



(2),(3)



(4),(5)



(4),(5)

Figure 5.9: Algorithm Stages

Organization of PBML Code

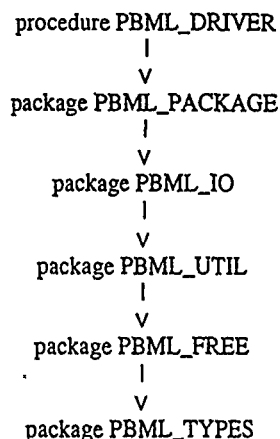


Figure 5.10: Compilation Dependencies

Search Constraints Common to All Versions

An exhaustive search approach to function decomposition becomes intractable for functions on more than two or three variables. Therefore, it is necessary to limit the search. There are some search limits that all the versions have in common. For a given partition of variables, the basic approach of each version is to compute the column multiplicity ν of the function with respect to that partition. The value of ν is then compared to a threshold that is version dependent. If the threshold is exceeded then this decomposition is dropped from further consideration. The idea is that we only want to pursue decompositions that are reducing cost. It is in how much of a reduction we want or how we measure cost that the versions differ. For our highest threshold (i.e. most exhaustive search) we know of no functions where a more exhaustive search would produce a lower overall cost decomposition. However, for our other thresholds we know that some desirable decompositions are being dropped. Using this threshold significantly reduces the search space; since otherwise every partition yields a decomposition whose children must be decomposed before we know that it will not result in an lower overall cost.

When the threshold is not exceeded and the decomposition is pursued, the first step is to form the children of the decomposition, F and ϕ . All the versions form F and ϕ by using the binary equivalent of an enumeration of the columns generated in counting up the column multiplicity. This simple approach to defining F and ϕ substantially reduces the overall search space. There are typically many different F 's and ϕ 's for a given decomposition. At least in some cases, how F and ϕ are defined affects the decomposability of F and ϕ and, consequently, the eventual cost of the overall decomposition. Therefore, to be exhaustive, we must assess the decomposability of all the different possible values for F and ϕ before arriving at a specific F

and ϕ . None of the AFD versions did this.

Search Constraint Differences Between Versions

There are two classes of variable partitions used by the AFD algorithms. Version 1 and all the 2 versions (i.e. 2, 2a, 2b, 2ab) partition the variables into two disjoint sets. One set is input into F and the other into ϕ . Version 3 and all the 4 (i.e. 4, 4a, 4b, 4ab) versions partition the variables into three disjoint sets. One set is input into F only, the second set is input into ϕ only, and the third set is input to both F and ϕ . We call this latter class of partitions the "shared variable" class while the first class is called "no-shared variables." When only no-shared variable partitions are used in a search it is possible for the specific values assigned to F and ϕ to affect the cost of the overall decomposition. We know of no cases where the method of assigning values to F and ϕ affects the overall decomposition when shared variable partitions are used in the search. It is possible that one has an alternative between searching through values for F and ϕ and searching with shared variable partitions; with either approach resulting in optimal decompositions.

The "a" versions (i.e. versions 2a, 2ab, 4a and 4ab) differ from the other versions in that they are "greedy." The "ε" versions search through partitions in order of increasing numbers of variables input into ϕ . When a cost saving decomposition is found for some number of input variables into ϕ then partitions with a larger number of variables input into ϕ are not considered. That is, once a decomposition is found for some number of variables into ϕ we pursue that decomposition but do not backup to consider larger numbers of inputs into ϕ . The idea here is that we want to break the original function into pieces that are as small as possible. Therefore, when we have succeeded in breaking out a small piece, we do not worry about trying to break out a larger piece. The "a" versions run substantially faster than the other versions and perform only slightly worse in terms of the cost of the decompositions produced.

As discussed previously, the AFD algorithms compare the cost of a candidate decomposition to a threshold. There are two methods for computing the cost of a candidate decomposition. The "b" versions (i.e. version 2b, 2ab, 4b, 4ab) compute cost based on the cardinality of the components (DPFC). The other versions compute cost based on the number of variables input into each component of the decomposition (DFC). That is,

$$DPFC = \sum_{p_i \in P} [p_i]$$

and

$$DFC = \sum_{p_i \in P} 2^{n_i}.$$

When the components are total functions the two costs are the same. However, when a component is a partial function, which can occur even when the input is a total function, DPFC is less than DFC. Since candidate decompositions are pursued whenever the cost is less than the threshold, "b" versions will in general conduct a larger search.

Version 1 : $NU_MAX = \min(NU_FEATURE, NU_LUB)$.
 Version 2 : $NU_MAX = \min(NU_NEG_ALL_TOTAL, NU_LUB)$.
 Version 2a : $NU_MAX = \min(NU_NEG_ALL_TOTAL, NU_LUB)$.
 Version 2b : $NU_MAX = \min(NU_NEG, NU_LUB)$.
 Version 2ab : $NU_MAX = \min(NU_NEG, NU_LUB)$.
 Version 3 : $NU_MAX = \min(NU_FEATURE, NU_LUB)$.
 Version 4 : $NU_MAX = \min(NU_NEG_ALL_TOTAL, NU_LUB)$.
 Version 4a : $NU_MAX = \min(NU_NEG_ALL_TOTAL, NU_LUB)$.
 Version 4b : $NU_MAX = \min(NU_NEG, NU_LUB)$.
 Version 4ab : $NU_MAX = \min(NU_NEG, NU_LUB)$.

Figure 5.11: NU_MAX for Each Version of the AFD Algorithm

Finally, the versions differ in the threshold they use to evaluate candidate decompositions. In essence, versions 1 and 3 set the threshold such that decompositions are pursued if they are “featured.” A decomposition is featured if ϕ has fewer output variables than input variables (c.f. [48, pp.64-66]). The “2” and “4” versions (i.e. versions 2, 2a, 2b, 2ab, 4, 4a, 4b, and 4ab) set the threshold such that decompositions are pursued if they result in a cost reduction (what we call a negative decomposition). In general there are many more featured decompositions than negative decompositions of a given functions. Therefore, the featured based versions are much more exhaustive than the other versions.

NU_MAX is defined in Figure 5.11. The various NU ’s are defined as follows where s is the number of inputs to ϕ only and R is the total number of variables input to ϕ (including shared variables).

- NU_LUB is the least upper bound on ν for a given partition of variables. NU_LUB is $\min(2^s, 2^{2^{n-R}})$.
- NU_NEG is the ν such that if $\nu > NU_NEG$ then no negative decomposition exists for the partition being considered. NU_NEG is the largest ν such that $[(\log(\nu))2^R + \nu 2^{n-s}] < 2^n$.
- $NU_NEG_INPUT_TOTAL$ is the ν such that if the input function is total then $\nu < NU_NEG_INPUT_TOTAL$ implies that a negative decomposition necessarily exists. This was used in the early b versions with unintended results. $NU_NEG_INPUT_TOTAL$ is the largest ν such that $[(\log(\nu))2^R + \nu 2^{n-s}] < [f]$.
- $NU_NEG_ALL_TOTAL$ is the ν such that if all functions involved are total then $\nu < NU_NEG_ALL_TOTAL$ implies that a negative decomposition necessarily exists. $NU_NEG_ALL_TOTAL$ is the largest ν such that $k2^{r-n} + 2^{k-s} < 1$.

- *NU_FEATURE* is the ν such that $\nu < \text{NU_FEATURE}$ implies that a featured decomposition necessarily exists. *NU_FEATURE* is 2^{s-1} .

NU_MAX is used to halt the counting of columns in determining column multiplicity and, when ν does reach *NU_MAX* we know that we do not want to pursue this partition of variables any further. However, in general, just because ν is less than *NU_MAX* does not ensure us that this partition will be a desired decomposition. Therefore, it is necessary to go ahead and form the decomposition (F and ϕ , but do not try to decompose F or ϕ) and determine its cost before deciding whether or not to include it in the current decomposition tree. We think it turns out that, except for the b versions, ν less than *NU_MAX* does ensure us of a desired decomposition.

The relationship between the search space of the different versions is $V_{2a} \subset V_2 \subset V_{2b} \subset V_1 \subset V_3$, $V_{4a} \subset V_4 \subset V_{4b} \subset V_3$ and $V_{2i} \subset V_{4i}$ for $i = a, b, ab$ or blank. Versions 2a and 4a were used for most of the experimental work described in Chapter 6. Unlike all the other versions, we found no functions that version 3 did not find the best known representation. Therefore, version 3 can not be ruled out as a possible optimal algorithm.

In summary, there are essentially three dimensions in the AFD version "space."

- 1. Greedy and 2. Not Greedy.
- 1. Not Shared and 2. Shared.
- 1. Negative DPFC, 2. Negative DFC and 3. Features required.

Thus, a point in this space (e.g. (1,2,1)) corresponds to a version of the AFD algorithm; in particular:

Version 1 is (2,1,3).

Version 2 is (2,1,1).

Version 2a is (1,1,1).

Version 2b is (2,1,2).

Version 2ab is (1,1,2).

Version 3 is (2,2,3).

Version 4 is (2,4,1).

Version 4a is (1,2,1).

Version 4b is (2,2,2).

Version 4ab is (1,2,2).

If version i has coordinates (a_i, b_i, c_i) , version j has coordinates (a_j, b_j, c_j) and $i \neq j$ then the search space of version j is a proper subset of the space of version i whenever

$$a_i \geq a_j, b_i \geq b_j, c_i \geq c_j.$$

For example, version 4b = (2,2,2) does a larger search than version 4 = (2,2,1). These relationships are summarized in Table 5.13. Note that we cannot draw conclusions from this about the relative size of the searches of some versions, e.g. version 2ab versus 4a. Note also that we did not implement versions corresponding to (1,1,3) or (1,2,3) because the feature based versions were so slow.

1. Not Shared:

	1. Neg DFC	2. Neg DPFC	3. Featured
1. Greedy	2a	2ab	-
2. Not Greedy	2	2b	1

2. Shared:

	1. Neg DFC	2. Neg DPFC	3. Featured
1. Greedy	4a	4ab	-
2. Not Greedy	4	4b	3

Table 5.13: AFD Algorithm Version Space

5.4 Ada Function Decomposition Program Performance

We are interested in how well the AFD algorithms decompose functions and in how long the decomposition takes. This section reports on the results of several experiments to assess the algorithms' performance.

We used the various versions of the Ada Function Decomposition program on VAX and MICROVAX computers to decompose well over 1000 different functions, ranging in size from 4 variables to 10 variables, ranging in cost complexity from 0 percent (most patterned) to 100 percent (completely unpatterned or 'random') and ranging in number of cares from 5 to 100 percent. Since many different experiments had been performed, there was adequate data to draw some conclusions about the relative performance of the different versions of the algorithm in terms of both cost reduction and run-time. Two subsets of data were extracted from the PT 1 data base and some statistical analysis was performed on them.

The first subset, 'Set A,' was composed of the output for all functions that had been decomposed by all ten of the versions of the program. There are 64 functions in this category. The second subset, 'Set B,' was composed of the output for all functions that had been decomposed by every version of the program with the exception of version 3. Using version 3 to decompose functions on six or more variables generated run-times that were far too great. Therefore, because Set A excluded all functions that were not decomposed using version 3, it necessarily excluded all functions on more than five variables (with the exception of two very simple functions). Set B was formed so that we could compare the other nine versions on functions of larger size. Set B contained 119 functions. Set A is a subset of Set B.

The first 14 functions in Set A were from a set of 'trick functions' that was constructed to test certain aspects of the various algorithms. This included the 'checker-board function' on five, six, seven and eight variables ; four different functions whose optimal decompositions included shared variables; and several functions that could

not be decomposed. The last 50 functions of Set A were all randomly generated functions on five variables.

Set B included all the functions in Set A. In addition, it included images of the letters R and A, and several test functions on six and seven variables that were designed either to decompose in certain irregular ways or to be non-decomposable. It also included a group of 50 random six-variable functions.

To summarize, the functions in Sets A and B ranged from the highly patterned checkerboard functions to the very unpatterned 'nodecomp' functions, but the great majority of them were unpatterned randomly generated functions; they ranged in size from four variables to eight variables but the majority of them were either five or six variable functions; and all of them without exception were total (i.e. none of them had any 'don't care' conditions). In addition to these sets where comparisons were made on average, there are some individual functions whose decomposition gives some insight into the algorithms' performance.

The following two sections consider the relative performance of the various versions in the individual areas of cost reduction and run-time.

5.4.1 Cost Reduction Performance

Set A and Set B Comparisons

Cost reduction is the primary aim of function decomposition. The driving purpose behind each of the versions of the program is to find a low cost representation of any given function, if one exists, and, hopefully, to find the lowest or "optimal" representation of the function. While we cannot guarantee that any decomposition found is truly optimal (except in a few cases that are amenable to theoretical analysis) the data that we collected have shown that all the versions of the program are able to find relatively low cost representations for most functions that are decomposable. For functions that are highly patterned and have theoretical optimal representations, all of the versions are able to find the optimal representations. For functions that are more complex, the different versions vary somewhat in performance with the more complete searches generally finding lower cost representations. The results of the Set A comparison bring out this difference primarily with respect to version 3 on five variables or less. The results on Set B show this difference on the other versions.

Version 3 may find optimal representations. No other version was ever able to find any representation of a function with a lower cost than was found by version 3, (although they were often able to find a representation with the same cost), nor were we able to decompose any functions by hand to a lower cost.

The algorithms that do not directly consider shared variables occasionally found a shared-variable decomposition through the creation of a new variable and a table of zero cost.

The versions' cost reduction performance is shown in Table 5.14. Probably the most important thing to note from this is that, in relative terms, the difference in cost reduction performance from the least exhaustive algorithm (version 2a) to the most

Version	Set A DFC	Set B DFC
3	15.93	NA
1	16.19	38.34
4	16.38	38.35
4a	16.38	38.35
4b	16.38	38.39
4ab	16.69	38.52
2b	17.00	38.69
2	17.06	38.72
2a	17.06	38.72
2ab	17.12	38.69

Table 5.14: Average DFC for Set A and Set B

exhaustive algorithm (version 3) is quite small. Other data suggested that version 2a was particularly likely to do nearly as well as version 3 when the function being decomposed was very patterned. Since most of the functions that were decomposed during the remainder of the phenomenology study fell into the highly patterned category, this result was one of the things that influenced us to use the 'greedy' algorithms.

We not only wanted to compare the decomposition performance between the versions, we also would like to know how close the algorithm was doing relative to the best possible decomposition. Some of the functions that we ran have well known representations (e.g. addition, parity, palindromes, functions with only one minority element). All AFD versions found the expected decompositions for these functions.

K-Clique Function Example

There were some cases where version 2a performed poorly. However, these were runs involving large functions ($n = 10$) where the algorithm was not allowed to run to completion. For example, the 3-clique function on a 5-node graph is a function with 10 variables. Version 2a was allowed to run on this function for about 170,000 seconds. The best decomposition found to that point had a DFC of 360 or 35.2 percent. Version 4a found a 164 DFC or 16.0 percent decomposition at 65,000 seconds. We know this function has a 116 DFC or 14.5 percent decomposition using Savage's sum-of-products form. Therefore, when the algorithms are not allowed to run to completion, version 4a can substantially outperform version 2a in a given amount of time.

Decomposition of Neural Net Like Functions

The AFD program tries to decompose functions by breaking out one piece at a time. We wonder whether or not there are some decompositions that cannot be found this way? Neural Nets have an architecture that has a low Decomposed Function

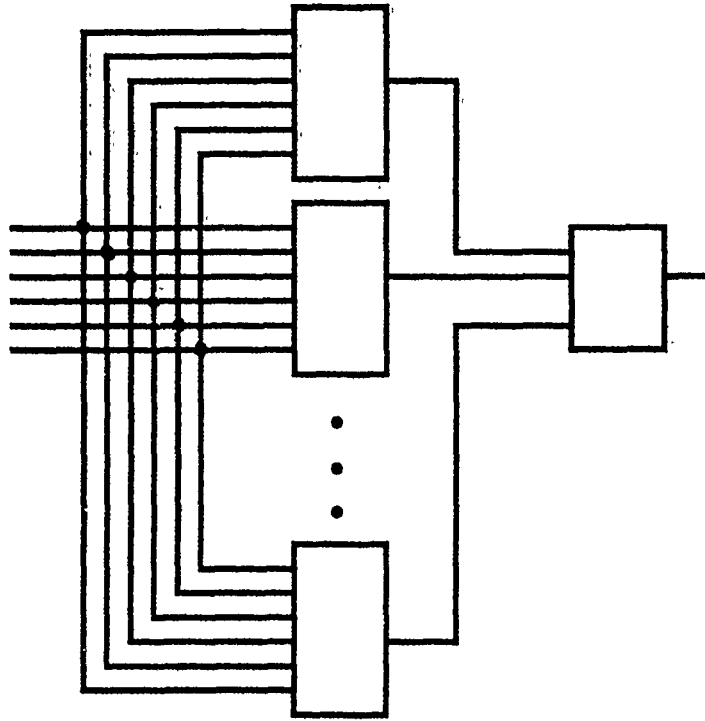


Figure 5.12: Neural Net Gross Architecture

Architecture	$[f]$	NN-DFC
6-2	64	44
8-2	256	60
8-4	256	124

Table 5.15: DFC of NN Like Architectures

Cardinality (DFC) but are not in a form that AFD would ever generate. Neural Nets have a gross architecture like Figure 5.12. Each of the boxes in the first layer have all n variables as input. Neural Nets have low computational complexity because the function in each box is very patterned. In real Neural Nets the function is a sum of products and perhaps a threshold. For comparison purposes we can let these boxes have a function of minimum cost (that is minimum among functions without vacuous variables). Each box in a Neural Net has an architecture like Figure 5.13. Therefore, each box has a DFC of $4(n - 1)$, where n is the number of input variables. Consider three specific Neural Net architectures. These architectures are identified as $a - b$, where a is the number of variables and b is the number of nodes in the first layer. Figure 5.14 shows the 6-2 architecture. The DFC's of these architectures are equal to $b(4(a - 1)) + 4(b - 1)$, see Table 5.15. Most other architectures on a small number of variables have a DFC which exceeds $[f]$. Although we know that the AFD program

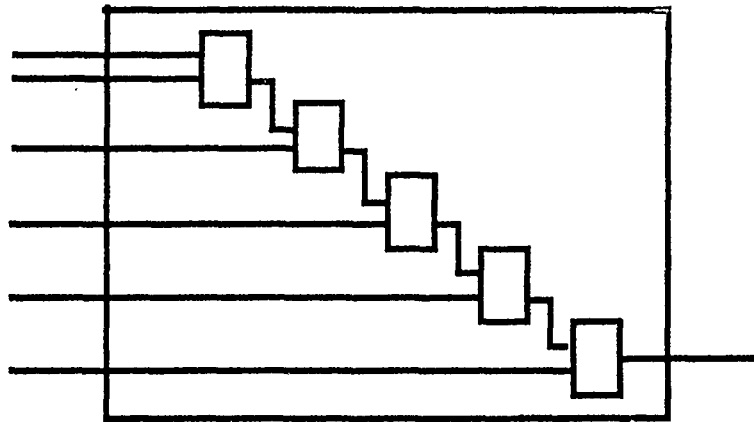


Figure 5.13: Detailed Architecture of a Neural Net Component

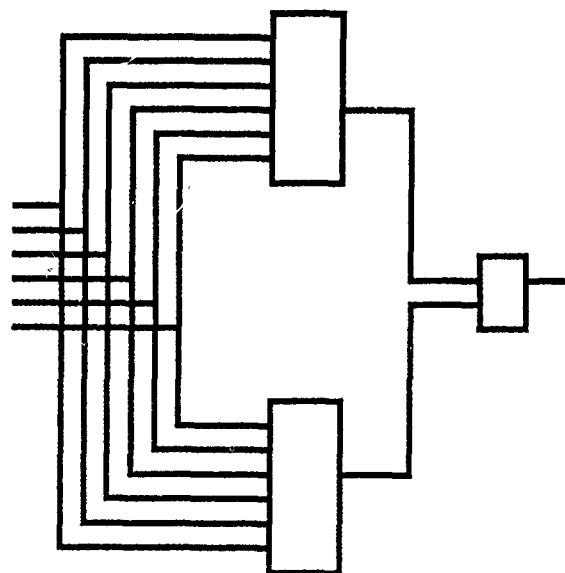


Figure 5.14: Specific NN Architectures

Architecture	[f]	NN-DFC	AFD-DFC	
			Maximum	Average
6-2	64	44	40	25.2
8-2	256	60	56	42.0
8-4	256	124	100	48.8

Table 5.16: AFD-DFC of NN Like Architectures

will not find the same architecture as the Neural Nets, we hope that it will find some other architecture with at most the same DFC. If it does not then we know that the AFD approach fails to recognize this class of patterns. To test this we generated 10 functions with each of the NN architectures. The function for each box was selected randomly from the 2-variable functions with DFC of 4 (i.e. constant and projection functions were excluded). These functions were then run on AFD version 2a. The results are as in Table 5.16. In summary, the DFC found by AFD was always less than that of the original NN architecture.

5.4.2 Run-Time Performance

Running on a Vax 11/780 with a throughput of roughly 1 million-instructions-per-second (MIPS), the different versions exhibited run-times ranging from less than a second up to more than 100,000 seconds. In order to design a reasonable number of experiments of a reasonable size, we needed an ability to estimate the run-time as a function of the number of variables, number of cares and number of minority elements, and version of the AFD program. This ability to estimate run-time allowed us to experiment with the best version of the AFD algorithm that time would allow.

The first step in our run-time analysis was to make comparisons of all algorithms on the Set A and Set B functions. Table 5.17 shows the average run-time comparisons for Set A and Set B. Run-time is measured in seconds of CPU time on our Vax 11/780. Although the cost reduction performances of all the versions were roughly equivalent, their run-times varied greatly.

Some versions always had lower run-times *and* decomposed functions to lower costs than certain other versions. For instance, version 2a runs faster than version 2ab and finds an equal or lower cost decomposition for every function in Set B. After getting rid of the versions that did not show any increase in cost reduction performance to justify their increase in run-time, we were left with the following versions, ranked from longest to shortest run-times: version 3, version 1, version 4a, version 2b and version 2a.

We searched the PT 1 data base and pulled out the functions on a given number of variables that were submitted to one of the five 'good' versions and recorded how many there were and their maximum, minimum and average run-times. There were two distinctions that we made in this process. First of all, we did not include run-time data

Version	Set A	Set B
2a	0.9	2.4
2	1.0	2.5
2ab	1.5	4.0
2b	2.0	4.9
4a	7.0	13.0
4ab	10.4	13.7
4	10.6	13.7
4b	14.0	16.0
1	70.0	275.0
3	220.0	NA

Table 5.17: Average Run-time for Set A and Set B

on functions with vacuous variables. The reason for this is that once the AFD program has eliminated one or more vacuous variables, it is then decomposing a function of one or more fewer variables. Secondly, we made a distinction between the run-times on the functions that did decompose and the ones that did not. The run-times of the functions that did not decompose all tended to be very closely grouped, while the functions that did decompose generated run-times that varied widely. Tables 5.18 and 5.19 show these results. The four entries for each version - number of variables combination are (from top to bottom): number of runs, minimum run-time, average run-time, maximum run-time. Note the expected exponential trend in increasing run-times for any given version on functions that do not decompose.

Several experiments were performed to assess the relationship between run-time, DFC and number of minority elements. The number of minority elements is the number of elements of a function that have output 1 or the number of elements that have output 0, whichever is smaller. These experiments were all carried out on version 2a.

Figure 5.15 shows the relationship between run-time and DFC for eight variables. We found no consistent pattern other than a general tendency for functions that decompose to have larger average and much larger maximum run-times.

It was found that one of the largest factors influencing run-time was the number of minority elements in a function, particularly when the number of minority elements was very small; therefore, functions were generated on four, five, six, seven, eight and nine variables which each had a fixed number of minority elements but were otherwise generated with no intended pattern. The results are shown in Figures 5.16 through 5.18. Functions with a proportionally small number of minority elements always decompose somewhat, and most of those that decompose have widely varying run-times. Once the number of minority elements increases past a critical point (around 15 percent of the cardinality of the function) then a randomly generated function will usually not decompose (or decompose very little) and the run-time associated with it

No. of Var	Ver. 3	Ver. 1	Ver. 4a	Ver. 2b	Ver. 2a
4	5	5	5	5	5
	10.6	2.4	1.3	0.5	0.5
	14.5	3.1	2.9	0.6	0.5
	22.0	3.5	3.3	0.7	0.7
5	5	5	5	5	5
	982.3	27.1	15.5	1.6	1.5
	1107.7	32.2	15.5	1.6	1.6
	1365.6	37.0	15.6	1.7	1.6
6		5	5	5	5
		380.2	73.1	5.7	5.5
		448.6	73.7	5.8	5.5
		536.8	74.3	5.9	5.5
7		3	5	5	5
		21952.9	340.4	21.6	21.1
		23858.2	342.4	22.5	21.2
		25763.1	343.8	23.2	21.4
8			5	5	5
			1534.4	87.4	80.2
			1538.4	92.5	80.9
			1541.7	94.8	81.6
9			5	5	5
			6882.3	352.8	324.2
			6960.0	370.6	330.3
			7049.7	383.6	336.2
10					1
					1375.0
					1375.0
					1375.0

Table 5.18: Run-times for Functions That Did Not Decompose

No. of Var	Ver. 3	Ver. 1	Ver. 4a	Ver. 2b	Ver. 2a
4	36	37	41	37	2
	15.6	2.4	1.2	0.7	0.6
	24.3	4.1	1.9	0.9	0.6
	43.9	5.4	6.7	2.3	0.7
5	18	32	21	17	17
	2113.6	42.6	15.7	2.0	1.6
	3177.4	73.0	106.0	5.7	2.2
	4700.9	108.6	326.3	17.8	7.0
6		19	26	14	118
		NA	NA	10.6	4.1
		1528.2	92.2	116.2	19.8
		2443.4	326.2	457.4	248.8
7			4	3	130
			NA	NA	8.0
			4245.6	48.3	1289.1
			16884.8	144.3	5415.0
8					101
					20.1
					666.5
					6293.0
9					20
					139.5
					6109.8
					39930.5

Table 5.19: Run-Times for Functions That Did Decompose

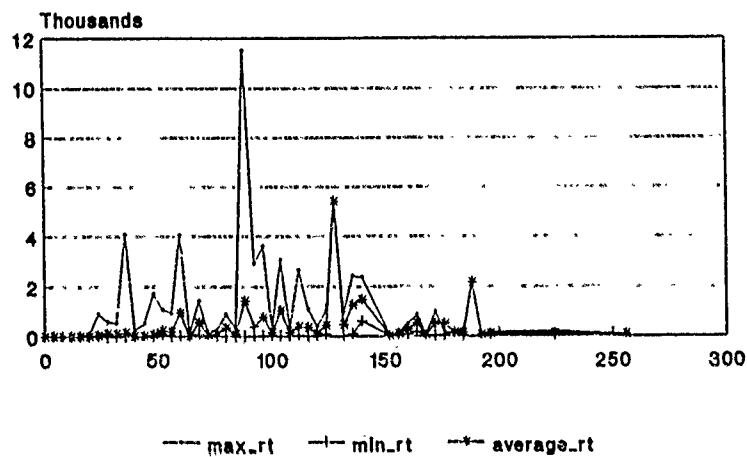


Figure 5.15: Run-time versus DFC for Functions on Eight Variables

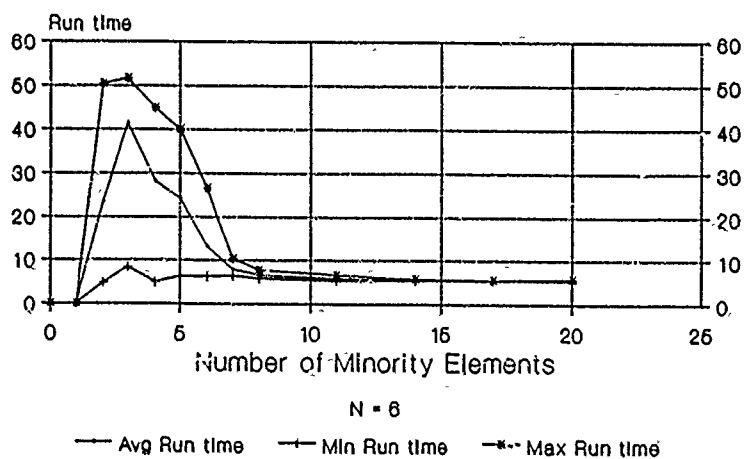


Figure 5.16: Run-Time versus Number of Minority Elements for Six Variable Functions

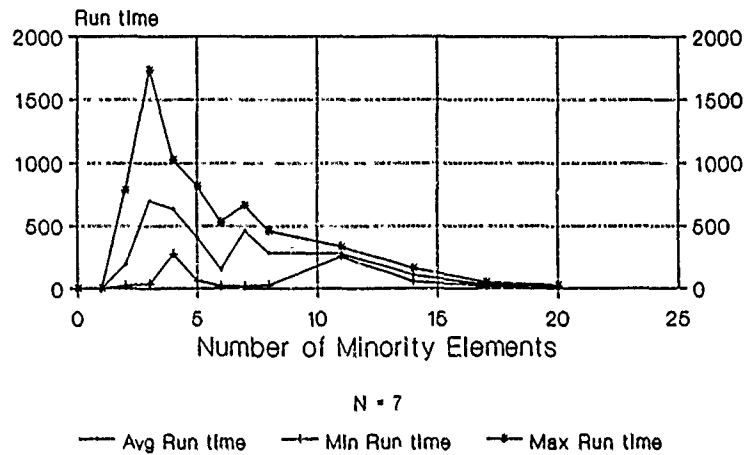


Figure 5.17: Run-Time versus Number of Minority Elements for Seven Variable Functions

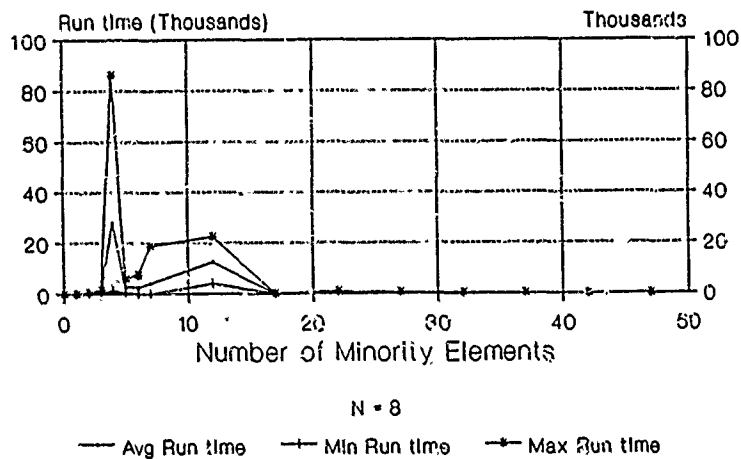


Figure 5.18: Run-Time versus Number of Minority Elements for Eight Variable Functions

will be very close to the expected run-time for any unpatterned function.

5.4.3 Summary

There was little difference in the average DFC performance between versions. There are cases though with substantial differences. There was a lot of difference in the average run-time between versions. We narrowed the number of versions to the five best, a set of versions where an increase in average run-time always corresponded to an increase in cost reduction performance. The experimental run-time data was consistent with the expected exponential relationship between run-time and the number of variables. There is a great deal of run-time sensitivity to the number of minority elements in a function. Even the faster versions had sufficient cost reduction performance to allow for some interesting results in the Pattern Phenomenology experiments (Chapter 6).

5.5 Summary

This chapter introduces the problem of function decomposition and discussed the Ada implementation that was used in this project. Approaches to decomposition consist of a test for decomposability and a search methodology. The test for decomposability is understood theoretically and is described in detail. The approaches to searching have little theoretical basis. We described several approaches that were experimented with in this project. We found that a 1 MIPS machine is capable of decomposing most functions on less than 10 binary variables in a matter of hours.

Chapter 6

Pattern Phenomenology

"...when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind ..."

- Lord Kelvin.

6.1 Introduction

This chapter has two objectives. On the one hand we experimentally test our assertion that Decomposed Function Cardinality (DFC) is a very general measure of pattern-ness. On the other hand, assuming that DFC is a general measure of pattern-ness, we begin sizing up the world relative to this new metric.

The first objective is complementary to the Chapter 4 theoretical demonstration of DFC's generality. In Chapter 4 we related DFC to the usual measures of computational complexity; information theoretic program length, algorithmic time complexity, and circuit size complexity.

This chapter reports pattern-ness measurements for many different kinds of things. These experiments are viewed in a way analogous to the first experiments with other scientific instruments. For example, the first uses of the mercury expansion thermometer were, at the same time, assessing the reasonableness of the temperature readings and quantifying for the first time various important temperatures (e.g. melting points). The analogy extends to the difference between perceived temperature and actual temperature. "Temperature" first became a concept based on the general sensation of warmth and cold. It only became a precise objective physical property after the invention of a thermometer. We now accept that any differences between sensed and measured temperatures are due to "thermal illusions" rather than some failing of thermometers. While we insist that a temperature measuring device reflect general trends in sensed temperature we expect situations where there are differences. When we are cold, things seem to be warmer than a thermometer would indicate. Therefore, a thermometer is not an exact predictor of sensed temperature. However, we would not accept a thermometer as a measure of temperature if it got too far

from our expectations. We think of DFC as a measure of pattern-ness much like a thermometer is a measure of temperature. We expect "pattern illusions" of two kinds. First there will be functions that are patterned whose pattern-ness is of a kind that we do not appreciate. Thinking now in terms of the pattern-ness of an image as compared to the DFC of the function that generates the image, some patterned functions (e.g. the prime number acceptor) will not look patterned to us. A second pattern illusion might result from our tendency to impose a certain degree of order on things. For example, people can see a face in almost any image with two horizontally displaced dark spots. As more specific examples, reference Figure 6.13 and 6.17; note that character 31 of font 3 looks patterned but does not have low DFC; character 48 of font 2 with permuted variables does not look patterned but has low DFC.

There are a few complicating factors in these experiments. For one thing we cannot be sure that the AFD program has found the true minimum cost. Most of the runs reported in this chapter were done with version 2a, the balance were done with version 4a. As indicated in Chapter 5 we do not think we are missing the optimum by very much, but there is certainly some bias on the measured DFC as compared to the true minimum DFC. Also, as discussed in Chapter 4, DFC does not include the costs of interconnections. Therefore, things that decompose but do so with high DFC may not be patterned at all. Finally, the AFD program's run-time is exponential in the number of input variables. Therefore, we are limited to measuring the DFC of functions with no more than about ten variables.

6.2 Randomly Generated Functions

6.2.1 Introduction

Patterned functions are both extremely rare and extremely common. If you look at all functions and choose a function at random, assuming all functions are equally likely, then the function almost certainly *will not be* patterned. Have you ever noticed that if you watch a television that is not receiving a signal that realistic images never seem to appear? If what we are seeing is a truly random image then any particular realistic image (e.g. John Wayne sitting on a horse) is just as probable as any particular random looking image. You can watch these random images for a long time and, although some strange psychological phenomena start to happen, you never see anything that is remotely realistic. Realistic images are patterned and patterns are extremely rare in the space of all functions. On the other hand if you pick realistic functions, such as functions with names (e.g. addition, sine, palindrome acceptor), real images, real sounds, etc. then the function almost certainly *will be* patterned. That is, patterns are extremely common in the real world. It seems to us that this is a physical property of the world, kind of like mass or any other physical property.

This section will assess the cost distribution (mean, maximum, minimum, and occasionally standard deviations) for arbitrary functions, functions with a specific

number of minority elements, and functions with a specific number of cares. This assessment will be based on theoretical and experimental results.

6.2.2 Completely Random Functions

This section is concerned with the cost of completely random functions. That is, the class of functions to be considered includes all functions and each function has the same probability of being chosen as any other function. Of course these functions are "random" only in the sense that generating a function using a random source of 0's and 1's is one way of generating a sample of this class. All the functions in this class are completely deterministic. That is, they all produce a specific output for a specific input. Later sections will limit class membership to those functions with a specific number of minority elements or a specific number of cares.

We need some initial results that will allow us to characterize the number of functions with respect to DFC.

Theorem 6.1 *The minimum DFC of a function is either 0, 2, or a sum of powers of 2 greater than 2.*

Proof:

DFC is a sum of powers of 2 by definition. If a minimum representation has cost greater than 2 and an individual component of cost 2 then we must have a situation where the component of cost 2 (p_1) is connected to another component (p_2). Therefore, p_2 could be redefined without increasing p_2 's cost such that p_1 is not required. The representation with the redefined p_2 would have lower cost than the original representation which contradicts the assumption that we began with a minimum cost representation. Therefore, the assumption that we can have a minimum cost representation with a component of cost 2 is false.

□

Theorem 6.2 *An integer n is the sum of powers of 2 greater than 2 if and only if n is evenly divisible by 4.*

Proof:

⇒

Assume that n is the sum of powers of 2 greater than 2, i.e. $n = \sum_{i=1}^k 2^{p_i}$ where each $p_i \geq 2$. Thus, $n = 4 \sum_{i=1}^k 2^{p_i-2}$ where $p_i - 2 \geq 0$. The sum is a whole number since each of the terms is a whole number. Thus, n is divisible by 4.

⇐

Assume that n is divisible by 4, i.e. there exists a whole number n' such that $n = 4n'$. Thus $n = \sum_{i=1}^{n'} 4$.

□

Theorem 6.3 *All functions must have DFC's of 0, 2, or multiples of 4.*

Proof:

Follows from Theorems 6.1 and 6.2.

□

Theorem 6.4 *The maximum number of non-vacuous variables (n') for a function with $DFC \geq 4$ is $n' = \frac{DFC}{4} + 1$.*

Proof:

It follows from Theorem 6.5 that $4(n' - 1)$ is the minimum cost for n' non-vacuous variables; that is $n' \leq \frac{DFC}{4} + 1$. The equality follows since $\frac{DFC}{4} + 1$ is always an integer by Theorem 6.2.

□

Now we consider the number of functions of a given cost:

- cost = 0: There are n projection functions and 2 constant function for a total of $n + 2$ functions of cost 0.
- cost = 2: There are n complements of the projection functions for a total of n functions of cost 2.
- cost = 4: There can only be two non-vacuous variables by Theorem 6.4. There are n choose 2 (or $n(n - 1)/2$) pairs of variables. Let $n' = 2$. There are $2^{2^{n'}}$ total functions on n' variables. Of these, $n' + 2$ have cost 0 and n' have cost 2. By Theorem 6.3 above, no functions have cost 1 or 3. Therefore, there are $2^{2^{n'}} - (n' + 2) - n'$ functions left of cost 4. Since $n' = 2$, $2^{2^{n'}} - (n' + 2) - n' = 10$. Therefore, there are $10n(n - 1)/2$ functions of cost 4.
- Cost = 8: There can only be three non-vacuous variables. There are n choose 3 (or $n(n - 1)(n - 2)/6$) triples of variables. Let $n' = 3$. There are $2^{2^{n'}}$ total functions on n' variables. Of these, $n' + 2$ have cost 0, n' have cost 2 and $10n'(n' - 1)/2$ functions of cost 4. By Theorem 6.3 above, no functions have cost 1, 3, 5, 6 or 7. Therefore, there are $2^{2^{n'}} - (n' + 2) - n' - 10n'(n' - 1)/2$ functions left of cost 8. Since $n' = 3$, $2^{2^{n'}} - (n' + 2) - n' - 10n'(n' - 1)/2 = 218$. Therefore, there are $218n(n - 1)(n - 2)/6$ functions of cost 8.
- Cost = 12: There can only be four non-vacuous variables. There are n choose 4 (or $n(n - 1)(n - 2)(n - 3)/24$) groups of variables. Let $n' = 4$. There are $2^{2^{n'}}$ total functions on n' variables. Of these, $n' + 2$ have cost 0, n' have cost 2, $10n'(n' - 1)/2$ functions of cost 4 and $218n'(n' - 1)(n' - 2)/6$ functions of cost 8. By Theorem 6.3 above, no functions have cost 1, 3, 5, 6, 7, 9, 10, 11, 13, 14, or 15. We do not know how many functions there are of cost 16, but we

DFC	Number of functions
0	6
2	4
4	60
8	872
12	5,794
16	58,800
Total	65,536
Average	15.53
Less than 16	10.28%

Table 6.1: Number of Functions for a Given DFC

believe there are 58800. This comes from the Pascal Function Decomposition Experiment where we decomposed virtually all functions on 4 variables. The number of functions of cost 0, 2, 4 and 8 corresponds exactly to that predicted above for $n=4$, which lends some credence to the 58800 figure. Therefore, there are $2^{2^{n'}} - (n' + 2) - n' - 10n'(n' - 1)/2 - 218n'(n' - 1)(n' - 2)/6 - 58800$ functions left of cost 12. Since $n' = 4$, $2^{2^{n'}} - (n' + 2) - n' - 10n'(n' - 1)/2 - 218n'(n' - 1)(n' - 2)/6 - 58800 = 5794$. Therefore, there are $5794n(n - 1)(n - 2)(n - 3)/24$ functions of cost 12.

This procedure does not work for cost = 16. We are only assured of five non-vacuous variables which allows for many costs (i.e. 0, 2, 4, 8, 12, 16, 20, 24, 28, and 32).

Virtually all functions on four variables were run on a Pascal function decomposition program similar to AFD version 2a. The results are shown in Table 6.1, where all DFC values not listed had zero functions. Note that the number of functions with costs zero through 12 matches exactly with the theoretical results. This makes us think that the algorithm finds optimal decompositions on functions of four variables.

In summary, there are $n + 2$ functions of cost 0, n functions of cost 2, $10n(n - 1)/2$ functions of cost 4, $218n(n - 1)(n - 2)/6$ functions of cost 8, and $5794n(n - 1)(n - 2)(n - 3)/24$ functions of cost 12. Figure 6.1 is a graph of these relationships for various n 's.

We used the AFD program to extend the theoretical results on DFC histograms. We generated 745 random functions on five variables and decomposed them with versions 1, 4a, 2a and 3. Because of limited computer resources version 3 was run only on the first 92. The combined results were: 711 functions had cost 32, 22 had cost 28 and 12 had cost 24. Based on this, we would estimate that zero percent of the functions on five variables have cost 0 through 20, $12/745 \times 100 = 1.6$ percent have cost 24, $22/745 \times 100 = 3.0$ percent have cost 28, and $711/745 \times 100 = 95.4$

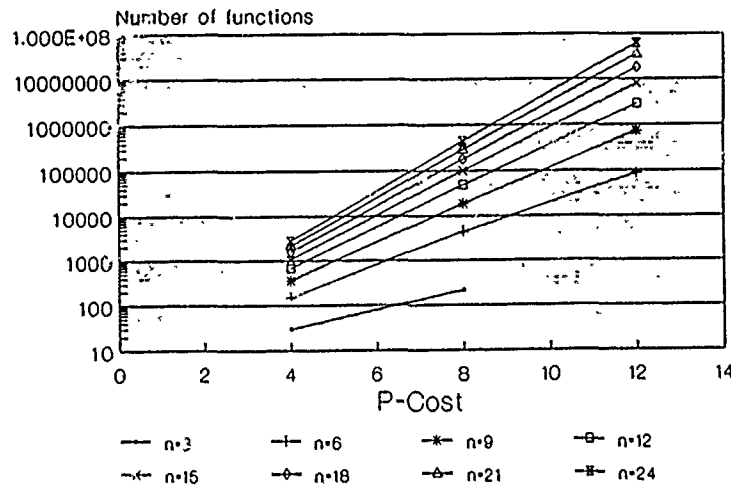


Figure 6.1: Number of Functions versus DFC for n up to 24

percent have cost 32. However, we must also consider that better decompositions may exist for many of the functions than we were able to find. Version 3 found no new decompositions beyond those found with versions 1 or 4a but did reduce the cost of one decomposition from 28 to 24. Based on this fact, we are suspicious that our estimated number of functions is inflated for a cost of 28. Figure 6.2 plots the estimated frequencies for costs of 24, 28 and 32 along with the theoretical frequencies developed earlier.

We generated 400 random functions on six variables; version 4a found no decompositions and version 1 found no decompositions in the first 100 of these. Therefore, we can be 98 percent confident that the fraction of "decomposable" functions on six variables is less than 1 percent. By "decomposable," we mean that the function has $DFC < 2^n$.

We generated 50 random functions on each of 7, 8, 9, and 10 variables, none of which decomposed. Version 4a was used for $n = 7, 8$, and 9 and version 2a for $n = 10$. Therefore, we can be 92 percent confident that the fraction of decomposable functions for each of these number of variables is less than five percent.

The number of functions with $L(e(r_f)) \leq l$ is less than or equal to 2^l by the information theoretic constraints (see Theorem A.17). This upper bound (i.e. 2^l) is fairly consistent with the trends in the experimental data and is perhaps a good estimate for the number of functions of cost less than or equal to l .

Functions on a very large number of variables ($n > 16$) will always decompose; at least a little bit. This result is from Lupanov'58 [34], see also Savage'76 [54, pp.116-120]. The Lupanov representation breaks a function into components. It is possible to realize these components with a cost savings if, instead of realizing the components independently, we first compute all possible minority elements and then use each minority element many times in computing the Lupanov components.

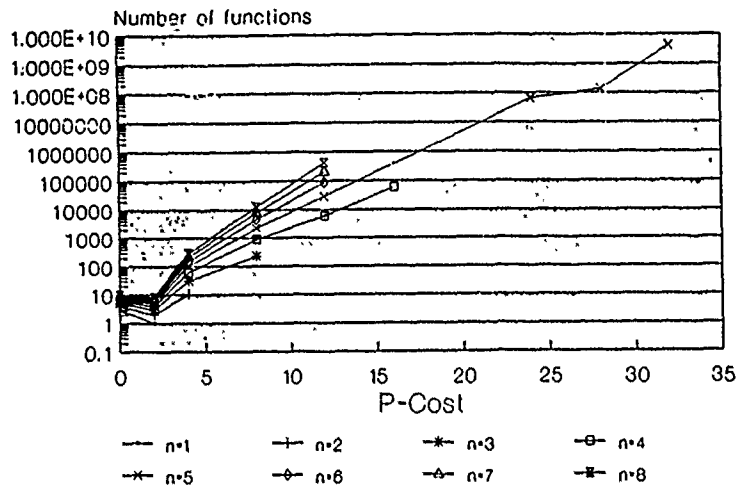


Figure 6.2: Number of Functions versus DFC for $n = 5$

The Lupanov representation has a complexity that grows as $2^n/n$. Based on rough calculations, the constants involved causes the Lupanov representation to be greater than 2^n for $n < 16$. Therefore, for the size of functions that we have experimented with (i.e. $n \leq 10$), this is not a factor. In terms of the general theory though, it is important. As discussed in Chapter 3, PT 1 is concerned with realizing a single function. We noted that when realizing multiple functions there would be some economy in re-using certain computations. However, for PT 1, we did not want to deal with this complication. Therefore, we defined the PT 1 problem to only involve a single function. The Lupanov upper bound makes it clear that even when realizing a single function, the fact that you generate multiple intermediate functions makes re-use a factor. Its not a terribly big factor; that is 2^n versus $2^n/n$, but a factor none the less. Of course the Information Theoretic constraints (Appendix A) still apply; therefore, if the cost measure included the cost of the interconnections then it is not a factor at all. In a sense, the Lupanov representation gets the DFC below 2^n by making the interconnection complexity very large.

In summary, for the vast majority of functions on five to ten variables, DFC is the same as the function's cardinality. Therefore, we expect randomly generated functions to have $DFC = 2^n$ and when the DFC is less than 2^n then there is something special about that function. Of course we think that "something special" is that it is patterned. In recognition of the Lupanov upper bound, this is not true for functions with more than 16 variables. When dealing with large functions, it may be desirable to reconsider our metric and possibly include interconnection costs.

6.2.3 Functions with a Specific Number of Minority Elements

Some functions have a definite minority in terms of output type. That is, a function may have 0 as its output for all but a small number of inputs. In this case we would say that 1 is the minority output type. An element of a function with output of the minority type is called a "minority element." This section will investigate the cost of functions with respect to their number of minority elements. Functions with a relatively small fraction of minority elements are common in practice. For example, the prime number acceptor only outputs a 1 for the relatively few numbers that are prime and a target detection algorithm would only output a 1 on the extremely small fraction of possible images that include targets.

There is a minimum cost for a function with a given number of non-vacuous variables. In particular, if a function has n non-vacuous variables then the cost of the function cannot be less than $4(n - 1)$. Therefore, if a function on n variables has i vacuous variables the cost of this function is at least $4(n - i - 1)$.

Theorem 6.5 *If F is the set of binary functions with n non-vacuous variables then the greatest lower bound on the cost of $f \in F$ is $4(n - 1)$.*

Proof:

There exists a function f in F with $n_i = 2$ for $i = 1, \dots, P$ where n_i is the number of input variables for component p_i of the representation of f and P is the total number of components in the representation. We are assured of f 's existence since for any representation with a p_i with $n_i \geq 3$ we could partition the variables of p_i into groups of size $n_i - 1$ and 1 with cost $2^{n_i-1} + 2^2$ which is less than 2^{n_i} for $n_i \geq 3$. The resulting function would be an element of F since the composition of functions which are essentially dependent on all their inputs is a function that is essentially dependent on all of its inputs. We can solve for P for such a representation since the total number of variables input to the p_i 's is the original n input variables plus the P variables generated by some p_i minus the final output variable which is not an input. That is,

$$\sum_{i=1}^P n_i = n + P - 1$$

, which reduces to $P = n - 1$ since all $n_i = 2$. The cost of this representation is:

$$\sum_{i=1}^P 2^{n_i} = \sum_{i=1}^{n-1} 2^2 = 4(n - 1).$$

This is a greatest lower bound since there exist functions with this cost, e.g. $x_1 + x_2 + \dots + x_n$, where $+$ is an OR operation and the x_i 's are Boolean variables.

□

There is a relationship between the number of minority elements in a function and the possible existence of vacuous variables. Intuitively, if a function has an odd

number of minority elements then it is not possible for a partition matrix to have two identical sets of columns and two identical sets of columns are possible for a function with a vacuous variable. We state this as a theorem.

Theorem 6.6 *Suppose F is the set of functions on n variables with exactly k minority elements, where k is an integer greater than zero. There exists $f \in F$ such that f has i vacuous variables if and only if k has 2^i as a factor.*

Proof:

\Rightarrow

f has k minority elements and i vacuous variables. If there are no vacuous variables then 2^0 is obviously a factor. Otherwise, choose a vacuous variable. The partition matrix with respect to this variable has two identical columns. Of course the columns contain the same number of minority elements and therefore the total function had an even number of minority elements. If we now drop this vacuous variable and repeat the argument for the remaining function we see that there is a factor of 2 in k for each vacuous variable.

\Leftarrow

Let $k = k'2^i$. We can construct an f with i vacuous variables one vacuous variable at a time. For the first vacuous variable, let $f(\dots, 0, \dots) = f(\dots, 1, \dots)$ but leave the specific values undefined. Now consider $f(\dots, 0, \dots)$ as the "function" and repeat the procedure i times. This is possible since there are a multiple of 2^i minority elements. The final "function" can be defined arbitrarily as long as it has k' minority elements. A function so constructed has i vacuous variables and $k'2^i$ minority elements.

□

This result could be used as a search constraint in the decomposition process. For example, if there are an odd number of minority elements then there is no point in testing for vacuous variables.

We now have a connection between the number of minority elements and the number of vacuous variables as well as between the number of vacuous variables and the minimum cost.

Theorem 6.7 *Let F be the set of binary functions on n variables with exactly k minority elements. Let i' be the largest i such that 2^i is a factor of k . Then for all $f \in F$ the cost of f is $\geq 4(n - i' - 1)$ and there exists an $f \in F$ such that the cost of f equals $4(n - i' - 1)$. That is, the cost of functions in F has a greatest lower bound of $4(n - i' - 1)$.*

Proof:

This follows from Theorem 6.5 and Theorem 6.6.

□

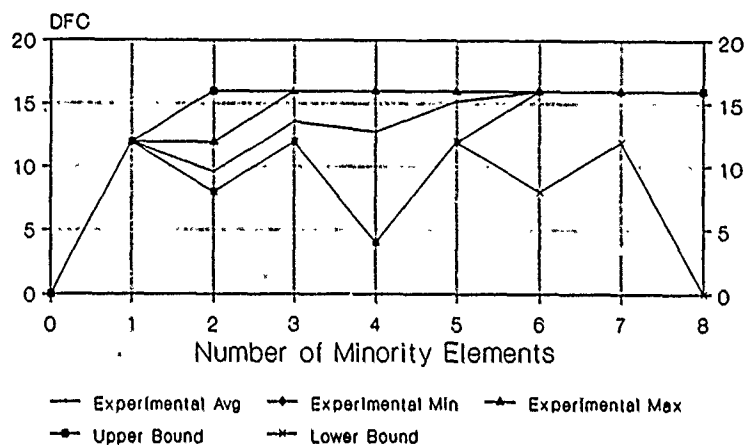


Figure 6.3: DFC With Respect to Number of Minority Elements, $n=4$

This lower bound on cost could be used as a stopping condition in a search for a best decomposition. That is, if the cost of the best decomposition so far is at the lower bound then the search can be stopped.

We have an upper bound on the cost of a function in terms of the number of minority elements. This upper bound is based on the fact that individual minority elements can be generated with a product and then summed together.

Theorem 6.8 *If f is a function on n variables with k minority elements then the cost of f is less than or equal to $4(nk - 1)$.*

To expand upon the theoretical bounds we generated a series of random functions with a controlled number of minority elements and determined their cost with the AFD program. We generated five functions each for 4, 6, 7, 8 and 9¹ input variables for each of the six different fractions of minority elements. On functions of five variables, we generated an average of about 500 functions for all possible number of minority elements (i.e. 0 - 15). The results of these experiments are plotted in Figures 6.3 through 6.7.

Table 6.2 lists the percentage of minority elements for a given fraction of cost. Based on this we might expect that the average cost of functions with less than 10 percent minority elements to be less than $0.9[f]$ and the average cost of functions with less than 5 percent minority elements to be less than $0.5[f]$.

6.2.4 Functions with a Specific Number of Don't Cares

A random total function was generated for each n from 4 to 10. These functions were then "sampled" and decomposed. By sampling we mean that we took a subset

¹Only four different nine variable functions were generated for each number of minority elements.

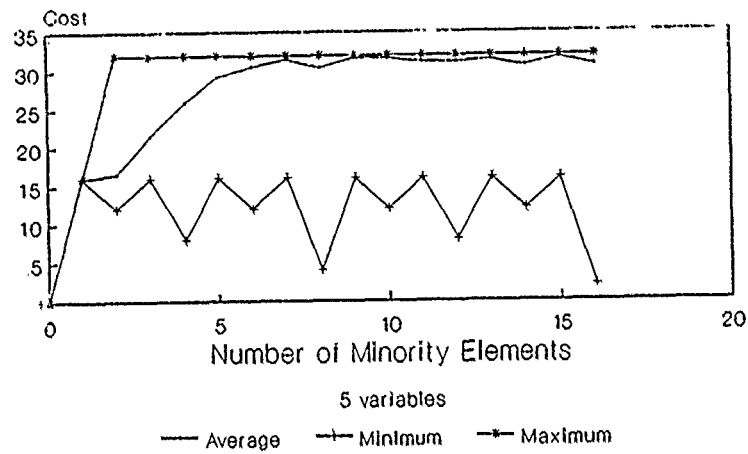


Figure 6.4: DFC With Respect to Number of Minority Elements, $n=5$

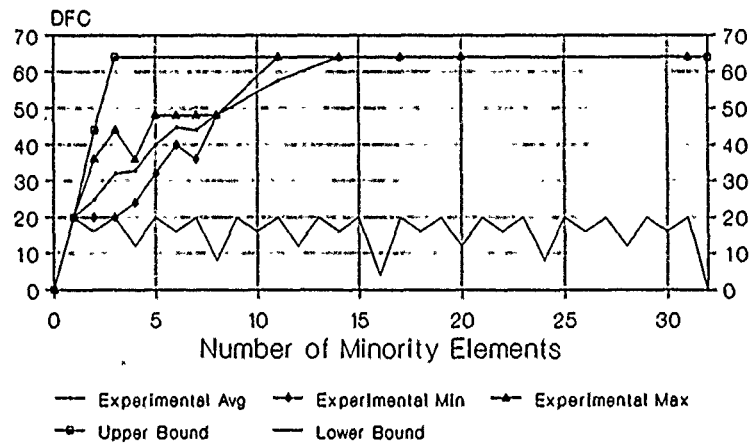


Figure 6.5: DFC With Respect to Number of Minority Elements, $n=6$

n	minority element fraction at $0.5[f]$	minority element fraction at $0.9[f]$
4	.06	.25
5	.03	.16
6	.05	.19
7	.06	.13
8	.07	.16

Table 6.2: Number of Minority Elements Required for a Given Cost

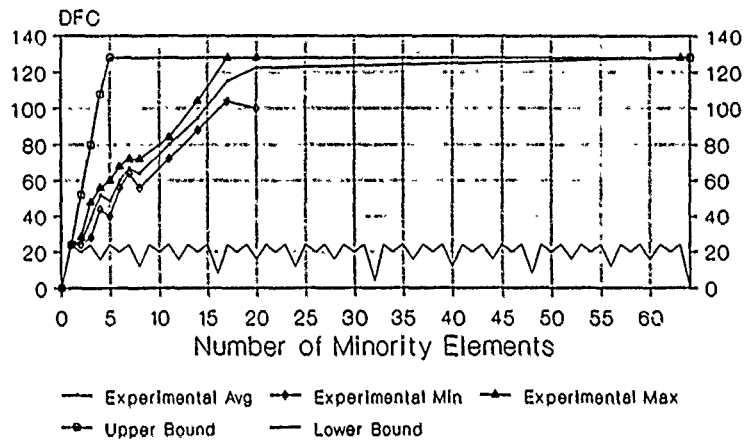


Figure 6.6: DFC With Respect to Number of Minority Elements, $n=7$

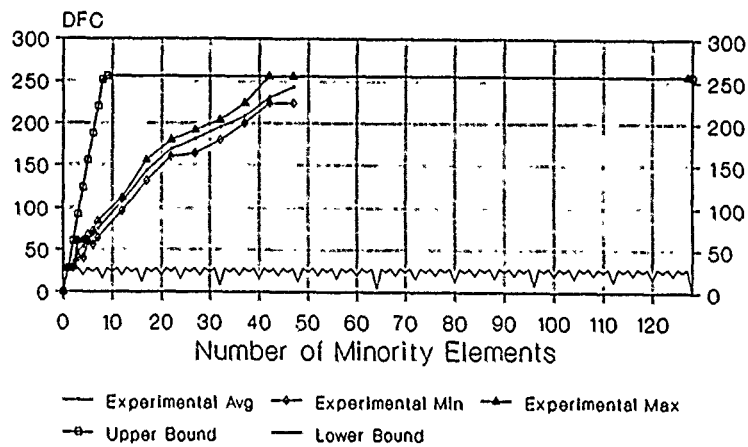


Figure 6.7: DFC With Respect to Number of Minority Elements, $n=8$

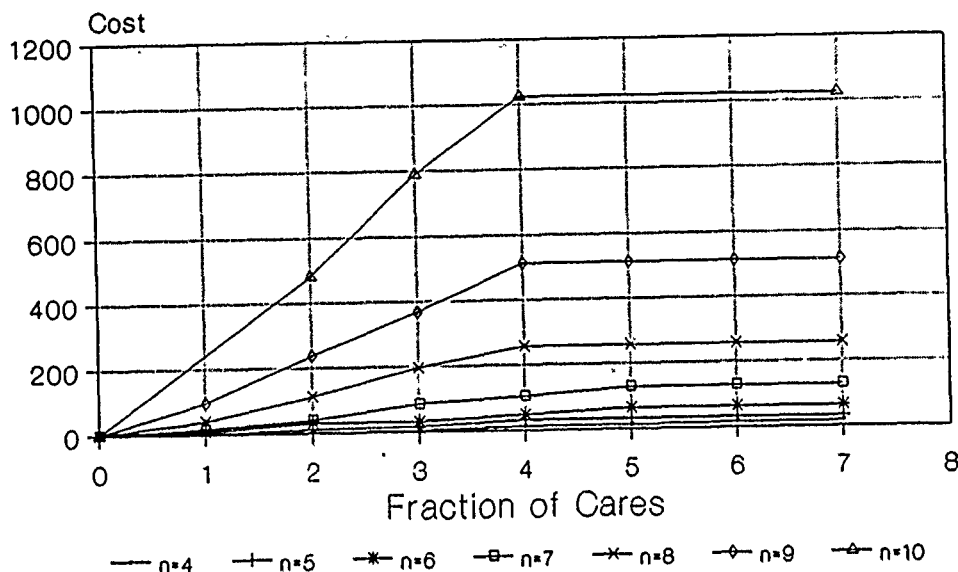


Figure 6.8: DFC as a Function of the Number of Cares

of the total function and then decomposed this partial function. The subsets were chosen randomly. In circuit design they often refer to the points not in a partial function as "Don't Cares" meaning that the output can be anything. Therefore, the size of the partial function is also called the number of "cares." The size of the subsets were 2, 5, 7, 10, 12 and 15 for $n = 4$. For all other n the samples sets contained $5 \times 2^{n-5} \times i$ elements, where $i = 1, 2, \dots, 6$. This corresponds to the following percentages: 15.6, 31.3, 46.9, 62.5, 78.1 and 93.8. Figure 6.8 plots the results of the AFD version 2a runs.

In order to assess the sensitivity of these results to the sample set, we repeated the above experiment five times for a random function on seven variables. That is, five different sample sets of each size were taken, but always from the same function. These results are shown in Figure 6.9. Note that DPFC can never exceed the number of cares. The DFC of a random function appears to increase linearly with the number of cares. The DFC of the partial functions reach the DFC of the total functions when the number of cares gets up to between 60 and 80 percent.

6.3 Non-randomly Generated Functions

In this section we go around with our new instrument and measure the pattern-ness of many different kinds of functions. We are testing the generality of our measure and exploring pattern-ness.

We identified several classes of functions that are small enough to be tested with the AFD program. These classes include numerical functions and sequences, symbolic functions, string manipulation functions, a graph theoretic function, images, and files.

Note that although we only report the DFC resulting from the AFD runs, when-

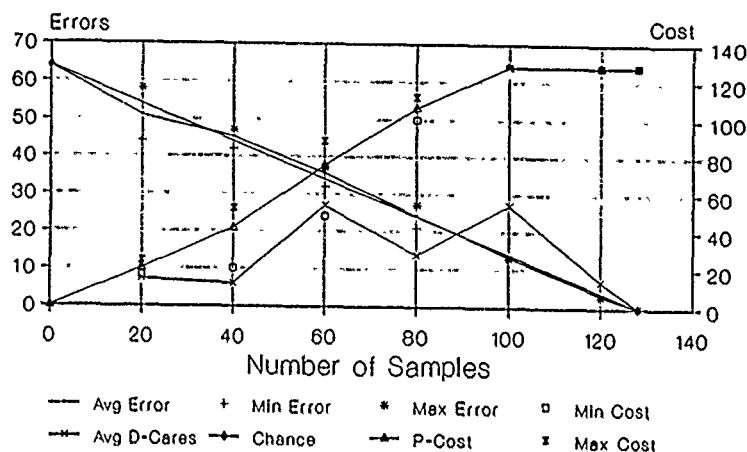


Figure 6.9: DFC as a Function of the Number of Cares, $n = 7$

ever the DFC is low the AFD program is also yielding an algorithm.

6.3.1 Numerical Functions and Sequences

By “numerical functions” we mean the usual arithmetic operations (addition, subtraction, multiplication and division), square and cube roots, trigonometric and logarithmic functions, prime and Fibonacci number acceptors, the greatest common divisor and the matrix determinant. The computation of numerical functions has a long history. The algorithms used to compute these functions have been hand-crafted over the centuries. To underscore the significance of what AFD does on these problems, try to imagine that we did not know how to compute addition and we were given the task of designing an addition algorithm. We might get out our algorithm design text, such as [3]; but what technique would we use? Are we going to try to represent the problem with a graph? Can we apply Heap Sort? What about dynamic programming? Where in this admittedly excellent text on algorithm design is there a method that would result in a good algorithm for addition? Of course the point is, algorithm design texts do not give methods for algorithm design, they give a tool kit of hand-made algorithms to choose from. Another point is that there are many problems, including some as simple as addition, that are not naturally constructed from things in the tool kit.

In this section we apply PT to a table defining addition and it produces the familiar digit-wise add and carry algorithm that we all learned in grade school. We believe that it is very important that this algorithm was produced by a *computer*, automatically, and the same computer program found the patterns in many other kinds of functions.

Numerical functions are represented as binary functions by using the usual binary

Output Bit	n	2^n	DFC	% DFC
1	8	256	4	1.6
2	8	256	12	4.7
3	8	256	20	7.8
4	8	256	28	10.9
5	8	256	28	10.9
Total		1280	92	7.2

Table 6.3: Addition.

equivalent of each input padded on the left with zeros to give the input the appropriate length. When there are two input numbers, as in addition, the binary function representation has as input the concatenation of the binary equivalents of the two input numbers. When the output is a non-binary number, we represent the numerical function by a binary function for each bit in the binary representation of the output. Consider addition as an example. Define the numerical function addition to be the sum of two numbers between 0 and 15. The output will be between 0 and 30. Each input number must be represented by four bits. The output must be represented by five bits. Therefore, we represent addition as five binary functions, each on eight variables. Although these 5 functions are decomposed independently, we sometimes refer to them as a single function. Numerical sequences are represented as "acceptors." For example, the Fibonacci sequence is represented as a numerical function whose output is 0 if the input is not in the Fibonacci sequence and whose output is 1 if the input is in the sequence. Tables 6.3 through 6.12 show the results for numerical functions.

Addition was represented by a binary function of the form

$$f : \{0,1\}^4 \times \{0,1\}^4 \rightarrow \{0,1\}^5.$$

Output bit 1, the least significant, is simply an exclusive OR of the least significant bits of the two inputs. The decomposition found for the more significant output bits is the familiar binary adder in combinational form. Notice that the more significant bits re-compute the less significant bits, so addition can be realized with a cost of 20 rather than 92. We have deliberately not tried to exploit this kind of savings (see Section 3.4.9). However, this suggests an approach of first decomposing one of multiple functions and then treating its output and intermediate states as inputs to a second function. In general, the DFC of adding two m -bit numbers is $16m + 8$. The cost of only the most significant bit is $8m - 4$.

Subtraction was represented by a binary function of the form

$$f : \{0,1\}^4 \times \{0,1\}^4 \rightarrow \{0,1\}^5$$

Output bit 5 is the sign bit and output bit 1 is the least significant of the 4 bits in the binary output.

Output Bit	n	2^n	DFC	% DFC
1	8	256	4	1.6
2	8	256	64	25.0
3	8	256	68	26.6
4	8	256	54	21.1
5	8	256	28	10.9
Total		1280	218	17.0

Table 6.4: Subtraction.

Output Bit	n	2^n	DFC	% DFC
1	4	16	4	25.0
2	4	16	12	75.0
3	4	16	12	75.0
4	4	16	12	75.0
Total		64	40	62.5
1	6	64	4	6.3
2	6	64	12	18.8
3	6	64	28	43.8
4	6	64	64	100.0
5	6	64	36	56.3
6	6	64	20	31.3
Total		384	164	42.7
1	8	256	4	1.6
2	8	256	12	4.7
3	8	256	28	10.9
4	8	256	112	43.8
5	8	256	256	100.0
6	8	256	256	100.0
7	8	256	168	65.6
8	8	256	56	21.9
Total		2048	892	43.6

Table 6.5: Multiplication.

Output Bit	n	2^n	DFC	% DFC
1	8	256	168	65.6
2	8	256	72	28.1
3	8	256	24	9.4
4	8	256	16	6.2
Total		1024	280	27.3

Table 6.6: Modulus.

Output Bit	n	2^n	DFC	% DFC
1	8	256	124	48.4
2	8	256	134	52.3
3	8	256	92	35.9
4	8	256	28	10.9
Total		1024	378	36.9

Table 6.7: Remainder.

Multiplication was represented by binary functions of the form

$$f : \{0, 1\}^{\frac{n}{2}} \times \{0, 1\}^{\frac{n}{2}} \rightarrow \{0, 1\}^n.$$

We evaluated functions with n equal to 4, 6, and 8. The output bits are listed in order of increasing significance. Note that for $n = 8$, output bits 5 and 6 did not decompose. One possible cause for this is that you need the results of computing the less significant bits before these bits are patterned. Another explanation might be that multiplication has a significant "buy-in" cost. That is, some patterned functions do not decompose until the number of variables gets above some threshold. The six-bit multiplication had a lower percentage DFC than the eight-bit. This is in part due to our using the better version 4a on six-bit multiplication. When six-bit multiplication is run on version 2a (which is what we had to use on eight-bit multiplication), the total DFC is 46.9 percent.

Modulus was represented by binary functions of the form

$$f : \{0, 1\}^4 \times \{0, 1\}^4 \rightarrow \{0, 1\}^8$$

where the output is the integer part of x_1 divided by x_2 .

Remainder was represented by binary functions of the form

$$f : \{0, 1\}^4 \times \{0, 1\}^4 \rightarrow \{0, 1\}^8$$

where the output is the integer part of the remainder from x_1 divided by x_2 .

Output Bit	n	2^n	DFC	% DFC
1	8	256	80	31.2
2	8	256	36	14.1
3	8	256	12	4.7
4	8	256	4	1.6
Total		1024	132	12.9

Table 6.8: Square Root.

Output Bit	n	2^n	DFC	% DFC
1	9	512	136	26.6
2	9	512	24	4.7
3	9	512	8	1.6
Total		1536	168	10.9

Table 6.9: Cube Root.

Square root was represented by a binary function of the form.

$$f : \{0,1\}^8 \rightarrow \{0,1\}^4.$$

The output is the binary equivalent of the integer part of the square root of the input.

The cube root was represented by a binary function of the form

$$f : \{0,1\}^9 \rightarrow \{0,1\}^3.$$

The output is the binary equivalent of the integer part of the cube root of the input.

The first quadrant of the sine function was represented by a binary function of the form

$$f : \{0,1\}^8 \rightarrow \{0,1\}^8.$$

An input of x degrees was encoded as the binary equivalent of the integer part of $255/90$ times x . The input varied from 0 to 90 degrees. The output is the binary equivalent of the integer part of $255 \sin x$. The output bits are listed in order of increasing significance.

The logarithm function was represented by a binary function of the form

$$f : \{0,1\}^8 \rightarrow \{0,1\}^3.$$

The output is the integer part of the logarithm to the base 2 of the input. The output bits are listed in order of increasing significance.

The "greater than" function is of the form

$$f : \{0,1\}^4 \times \{0,1\}^4 \rightarrow \{0,1\},$$

Output Bit	n	2^n	DFC	% DFC
1	8	256	256	100.0
2	8	256	256	100.0
3	8	256	256	100.0
4	8	256	224	87.5
5	8	256	176	68.8
6	8	256	104	40.6
7	8	256	54	21.1
8	8	256	24	9.4
Total		2048	1350	65.9

Table 6.10: Sine.

Output Bit	n	2^n	DFC	% DFC
1	8	256	24	9.4
2	8	256	20	7.8
3	8	256	12	4.7
Total		768	56	7.3

Table 6.11: Logarithm.

Function	n	2^n	DFC	% DFC
Greater Than	8	256	28	10.9
Factorial	5	32	28	87.5

Table 6.12: Miscellaneous Numerical Functions.

n	2^n	DFC	% DFC
6	64	64	100.0
7	128	104	81.2
8	256	196	76.6
9	512	336	65.6
10	1024	600	58.6

Table 6.13: Primality Tests.

n	2^n	DFC	% DFC
5	32	24	75.0
6	64	48	75.0
7	128	76	59.4
8	256	108	42.2
9	512	144	28.1

Table 6.14: Fibonacci Numbers.

where the output is 1 if the first 4 bit input is greater than the second 4 bit input and 0 otherwise.

The "factorial" function is of the form

$$f : \{0,1\}^5 \rightarrow \{0,1\},$$

where the output is 1 if and only if the input is some factorial (i.e. 1, 2, 6 or 24).

The prime number acceptors are functions of the form

$$f : \{0,1\}^n \rightarrow \{0,1\},$$

where the output is 1 if the input is a prime number and 0 otherwise (zero and one were considered prime). Runs were made with n ranging from 6 to 10. This is an example where n must be large before the function begins to decompose.

The Fibonacci number acceptors are functions of the form

$$f : \{0,1\}^n \rightarrow \{0,1\},$$

where the output is 1 if the input is a Fibonacci number and 0 otherwise. Runs were made with n ranging from 5 to 9.

One experiment concerned the patterns in deciding whether or not a binomial coefficient is an odd number. About 100 years ago, E. Lucas discovered that a choose b is odd if and only if every bit in b implies its corresponding bit in a . This is a highly patterned computation according to Pattern Theory and the AFD program

Function	n	2^n	DFC	% DFC
Lucas Function	6	64	20	7.8
Lucas Function	8	256	28	10.9

Table 6.15: DFC of Lucas Functions.

Function	n	2^n	DFC	% DFC
Binomial Coeff.	6	64	12	4.7
Binomial Coeff.	8	256	20	7.8

Table 6.16: DFC of Binomial Coefficient Based Functions.

rediscovered this pattern. Tables 6.15 and 6.16 summarize these results. The binomial coefficient functions are of the form

$$f : \{0,1\}^m \times \{0,1\}^m \rightarrow \{0,1\},$$

m is 3 or 4 and

$$f(a,b) = \begin{cases} 1 & \text{if } a \text{ choose } b \text{ is odd} \\ 0 & \text{if } a \text{ choose } b \text{ is even} \\ \text{don't care} & \text{if } a \text{ choose } b \text{ is undefined (i.e. } b > a) \end{cases}$$

The Lucas functions are of the form

$$g : \{0,1\}^m \times \{0,1\}^m \rightarrow \{0,1\},$$

m is 3 or 4.

$$g(a,b) = \begin{cases} 1 & \text{if each bit in } b \text{ implies its corresponding bit in } a \\ 0 & \text{otherwise} \end{cases}$$

Lucas's Theorem [12, p.2] says that $g(a,b) = f(a,b)$ when f is defined. Note that g has by definition a decomposition of minimum cost for no vacuous variables (i.e. $DFC(g) = 4(n-1)$).

The Greatest Common Divisor (GCD) problem is represented by a function of the form

$$f : \{0,1\}^4 \times \{0,1\}^4 \rightarrow \{0,1\}^4,$$

Since we are not interested in the GCD when either number is zero, 0000 represents decimal 1, 0001 represents decimal 2, ..., 1111 represents decimal 16. Output bit 1 is the most significant.

The determinant function is represented by a function of the form

$$f : \{0,1\}^9 \rightarrow \{0,1\}^3.$$

Output Bit	n	2^n	DFC	% DFC
1	8	256	48	18.8
2	8	256	172	67.2
3	8	256	164	64.1
4	8	256	4	1.6
Total		1024	388	37.9

Table 6.17: DFC of Greatest Common Divisor Function.

Output Bit	n	2^n	DFC	% DFC
zero/non-zero	9	512	128	25.0
positive/negative	9	512	108	21.1
± 1 or ± 2	9	512	34	6.6
Total		1536	270	17.6

Table 6.18: DFC of the Determinant Function.

The input is a 3×3 binary matrix. The first output bit indicates whether or not the determinant is zero. The second output bit indicates whether or not a non-zero determinant is positive or negative. The third output bit indicates whether the absolute value of a non-zero determinant is 1 or 2. Note that the size of the determinant function is $\log_5 2^9 = 1188.83$ since there are only five possible outputs (i.e. $-2, -1, 0, 1, 2$).

In summary, about 76 numeric functions were run on the AFD program. All decomposed except for 3 of the 18 bits associated with multiplication, 3 of the 8 sine bits and the prime number acceptor on 6 variables. Note that when the same function was done on different numbers of variables (such as multiplication, primality test and the Fibonacci number acceptor), the percentage cost went down as the number of variables went up. The binomial coefficient based functions were exceptions. Also note that the number of bits that do not decompose for multiplication was increasing with the number of variables.

6.3.2 Language Acceptors

We were encouraged by the ability of DFC to reflect the patterns in a wide variety of numerical functions. In this section, we generate "languages" to see if the DFC of a completely different kind of pattern is also low.

We did a series of experiments on a class of functions called "language acceptors." An abstract language is simply a subset of a set of strings. A language acceptor is a function which outputs 1 if the input string is in the language and outputs 0 if the input string is not in the language. There is a one-to-one correspondence between

total binary functions on strings and languages. An arbitrary language is not different from a random function; therefore, we would not expect an arbitrary language to correspond to a function that decomposes. We are interested in languages that can be generated with a formally defined grammar. A grammar is a set of rules. Each rule has a left part and a right part; both parts are strings. An element in a language is generated by starting with a special starting symbol and then replacing symbols that are the left part of some rule in the grammar with the right part of that same rule. If there exists some sequence of rule applications that will yield a given string from the starting symbol then that string is in the language. See [22] for a concise introduction to formal languages. A context-free grammar is one in which the left parts of the rules consist of a single symbol. In [22, pp.178-179] there are algorithms for accepting languages defined by context-free grammars in cubic time. If a language is defined by a context-free grammar then we would expect the language acceptor for that language to be patterned. Therefore, whether or not language acceptors decompose is a test of the generality of the DFC measure of pattern-ness.

To conduct a test of the DFC measure relative to language acceptance we generated two programs. One randomly generates a set of context-free syntactic rules, i.e. a context-free grammar. The grammars used in this experiment were generated and then edited. The minor editing was necessary to remove duplicate rules and generally ensure that a non-trivial language resulted. The second program takes a grammar as input and generates the corresponding language accepting function in AFD input format. Although languages generally include strings of any length, we defined the language acceptor function only for input strings of a fixed length (in particular, lengths of 9 bits). This was necessary because the AFD program is designed for functions defined on vectors (which are the same as fixed length strings). Reference Appendix A for more on the relationship between functions on strings and functions on vectors. A sample of the languages resulting from these software tools is shown in Table 6.19.

The languages were then decomposed with version 2a of the AFD program. The results of this experiment are in Table 6.20. In summary, 14 context-free language acceptors were generated and decomposed. The highest cost was about 25 percent and the average cost was less than 10 percent. This result supports the contention that DFC measures the pattern-ness of syntactically patterned functions.

6.3.3 String Manipulation Functions

In this section we generate functions with yet another class of patterns. These functions are most easily thought of as functions on binary strings. The "palindrome" function outputs 1 if the input binary string is symmetric about its center and outputs 0 otherwise. The "majority gate" function outputs 1 if the binary input contains more ones than zeros and outputs 0 otherwise. The "counting four ones" function outputs a 1 if and only if the input binary string contains exactly four ones. The "parity" function outputs 1 if and only if the input binary string has an odd number of ones. The "XOR" is the exclusive OR function where the output is 1 unless the

Language 13	Language 4	Language 2	Language 11
aaaaaaaaab	babbbbbba	ababababa	aaaaaaaaa
aaaaaabab	babbbbbbb	abababbab	aaaaabbbb
aaaaabaab	bbabbbbba	abababbba	aaaabbbba
aaaabaaab	bbabbbbbbb	ababbabab	aabaaabbb
aaaababab	bbbabbbbba	ababbabba	aababaabb
aaabaaaab	bbbabbbbb	ababbababa	aabababab
aaabaabab	bbbabbabba	ababbbbab	aabababba
aaababaab	bbbabbbbb	ababbbbba	aababbaaa
aabaaaaab	bbbbbabba	abbababab	aabbaaaaa
aabaaabab	bbbbbabbb	abbababba	abaaaaaaa
aabaabaab	bbbbbbaba	abbabbaba	abaaabbba
aababaaab	bbbbbbabb	abbabbbab	ababaabba
aabababab	bbbbbbbaa	abbabbbba	ababababa
baaaaaaab	bbbbbbabb	abbbababa	abababbba
baaaaabab	bbbbbbbaa	abbbabbab	ababbaaaa
baaaabaab	bbbbbbbaa	abbbabbba	abbaaaaaa
baaaabaaab		abbbbabab	baaaaaaaa
baaababab		abbbbabba	
baabaaaab		abbbbababa	
baabaabab		abbbbbbab	
baababaab		abbbbbbbba	
babaaaaab			
babaaabab			
babaabaab			
bababaaab			
babababab			
bbbbbbba			

Table 6.19: Sample Languages.

Language	n	2^n	DFC	% DFC	No. of Productions	Non-terminals
1	9	512	28	5.5	5	1
2	9	512	48	9.4	6	2
3	9	512	32	6.3	7	2
4	9	512	52	10.2	8	2
5	9	512	0	0	8	2
6	9	512	4	0.8	9	2
7	9	512	32	6.3	6	3
8	9	512	20	3.9	10	3
9	9	512	32	6.3	10	3
10	9	512	116	22.7	11	3
11	9	512	124	24.2	12	3
12	9	512	28	5.5	11	4
13	9	512	64	12.5	15	5
14	9	512	72	14.1	16	5
Average			46.6	9.1		

Table 6.20: DFC of Language Acceptors.

Function	n	2^n	DFC	% DFC
Palindrome	8	256	28	10.9
Majority Gate	7	128	48	37.5
Majority Gate	9	512	96	18.8
Counting Four Ones	7	128	64	50.0
Parity	7	128	24	18.8
Parity	8	256	28	10.9
Parity	9	512	32	6.3
XOR	7	128	24	18.8

Table 6.21: Miscellaneous String Manipulation Functions.

Output Bit	n	2^n	DFC	% DFC
1	8	256	28	10.9
2	8	256	60	23.4
3	8	256	68	26.6
4	8	256	68	26.6
5	8	256	68	26.6
6	8	256	68	26.6
7	8	256	60	23.4
8	8	256	28	10.9
Total		2048	448	21.9

Table 6.22: Sorting Eight 1-Bit Numbers.

Output Bit	n	2^n	DFC	% DFC
1	8	256	12	4.7
2	8	256	56	21.9
3	8	256	16	6.3
4	8	256	160	62.5
5	8	256	16	6.3
6	8	256	184	71.9
7	8	256	12	4.7
8	8	256	56	21.9
Total		2048	512	25.0

Table 6.23: DFC of Sorting Four 2-Bit Numbers.

input is all zeros or all ones. The results of the decomposition of these functions are shown in Table 6.21.

We did two "sorting" experiments. One considers sorting eight 1-bit numbers and the other considers sorting four 2-bit numbers. Note that when sorting one-bit numbers, $bit_i(x) = 1 - bit_{9-i}(255 - x)$. Also, for sorting four 2-bit numbers, the higher order bits of the output (i.e. bits 1, 3, 5, 7) are independent of the low order bits of the input (i.e. bits 2, 4, 6, 8). That is, inputs 2, 4, 6, and 8 are vacuous in bits 1, 3, 5, and 7.

In summary, six different string based functions were considered (palindromes, majority gate, counting, parity, exclusive OR, and sorting). A total of 24 binary functions derived from these were run on AFD version 2a. All 24 functions decomposed. This supports the contention that DFC reflects the patterns in string based functions.

Bit	Arc
1	1,2
2	1,3
3	1,4
4	1,5
5	2,3
6	2,4
7	2,5
8	3,4
9	3,5
10	4,5

Table 6.24: Input Bits Represent Arcs

6.3.4 A Graph Theoretic Function

This experiment concerns the patterns in deciding whether or not a graph has a k -clique. A graph has a k -clique if it has k nodes that are completely connected (i.e. each of the k nodes has an arc to all the other nodes in the clique). The k -clique problem is NP-complete [54, p.4]. Therefore we would not expect this function to be highly patterned.

The functions used in this experiment are of the form $f : \{0,1\}^{10} \rightarrow \{0,1\}$. The input is the representation of an undirected graph with five nodes. Each bit in the input indicates whether or not a given arc is in the graph as in Table 6.24. For example, there is an arc between nodes 1 and 2 if bit 1 of the input is 1. Therefore, the 10 input bits represent a graph and the output of the k -clique function is 1 if the graph has a k -clique and 0 otherwise. We will consider each of the k -clique functions for k 's of 1 through 5.

We cannot do the 1-clique problem with this set up since we do not allow for arcs from a node to itself. However, this limitation does not affect the other k -clique functions since such arcs do not change whether or not a graph has a k -clique (for $k > 1$). That is, if we had defined $g : \{0,1\}^{15} \rightarrow \{0,1\}$ and bits 1-10 are defined as in Table 6.24 and with bits 11-15 defined as in Table 6.25 then variables 11-15 would be vacuous for k -clique functions with $k > 1$.

The 2-clique function has only one minority element (i.e. a graph has a 2-clique unless it has no arcs), therefore has cost $4(n-1) = 36$. Similarly, the 5-clique function has cost 36.

We attempted to decompose the 3 and 4-clique functions with the AFD program, but did not find a decomposition as good as the Savage sum-of-products form (the AFD runs had not finished after 3 days). The best known decompositions then are as in Table 6.26.

The complete k -clique function (i.e. the input includes k and the graph) could be

Bit	Arc
11	1,1
12	2,2
13	3,3
14	4,4
15	5,5

Table 6.25: Additional Input Bits for Arcs to Self

k	DFC	Percentage DFC
1	0	0.0
2	36	3.5
3	116	11.3
4	116	11.3
5	36	3.5
Total	304	5.9

Table 6.26: DFC of the Various k -clique Functions on a Graph With 5 Nodes

computed from 1-clique, 2-clique, ... as follows. Say k is input as a 3-bit number; have five circuits, one that is true only if $k = 1$, one that is true only if $k = 2$, etc. (called a binary-to-positional transformation). The i^{th} of these circuits is then AND'ed with the i -clique function. The result of all the AND's is then OR'ed together. The result is the k -clique function. The realization of k -clique just described has the cost of the individual i -clique functions (304), plus the cost of the five counters ($5 \times 8 = 40$), plus the cost of five AND's ($5 \times 4 = 20$) and four OR's ($4 \times 4 = 16$). Therefore, the total cost of the k -clique function of the form $f : \{0, 1\}^3 \times \{0, 1\}^{10} \rightarrow \{0, 1\}$ is 380. The size of the function is 2^{13} , so as a percentage the DFC is 4.6 percent.

From [54, p.5], we see that the k -clique function on a graph with four nodes has the form $f : \{0, 1\}^2 \times \{0, 1\}^6 \rightarrow \{0, 1\}$. The 3-clique function on a graph with four nodes has $DFC = 44$. As in the k -clique function on 5-node graphs, the k -clique function on 4-node graphs could be computed with the above functions, four counters (cost $= 4 \times 4 = 16$), four AND's (cost $= 4 \times 4 = 16$), and three OR's (cost $= 3 \times 4 = 12$). Therefore, the total cost of the k -clique function on a 4-node graph is 128. The size of the function is 2^8 , so the percentage DFC is 50 percent.

In summary, the DFC of the k -clique function on 4-node graphs (input size 8-bits) is about 128 and on 5-node graphs (input size 13-bits) is about 380. Despite the fact that the complexity of the k -clique function grows rapidly (i.e. NP) as the size of the input increases, the complexity is low for a 5-node graph. It would be interesting to evaluate the DFC of several NP-Complete problems for several input sizes each. We

k	DFC	Percentage DFC
1	0	0.0
2	20	31.3
3	44	68.8
4	20	31.3
Total	84	32.8

Table 6.27: DFC of the Various k -clique Functions on a Graph With Four Nodes

Font Number	Font Name	Number of Characters
0	default	253
1	triplex	94
2	small	94
3	sans serif	95
4	Gothic	95

Table 6.28: Turbo Pascal V5.5 Font Sets

only have two data points on this single NP-Complete problem, but the growth in complexity is much less than we expected. There are many long-standing unanswered questions about the complexity of NP problems and decomposition may be an avenue to some new insights. In any case, DFC reflects the patterns in yet another context.

6.3.5 Images as Functions

In the preceding sections we saw that DFC captures the essential complexity of many kinds of functions. In this section, we consider the pattern-ness of images. Since DFC measures the pattern-ness of binary functions, we need to represent an image as a binary function. Suppose we want to assess the pattern-ness of a 16 pixel by 16 pixel black and white image. We represent this image as a binary function of the form

$$f : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\},$$

where $n = 4$ and $m = 4$. The first 4 bits of the input specify a column of the image, the last 4 bits specify a row and the output is the color (0 for white and 1 for black) of the pixel at that row and column.

This experiment uses a 16 by 16 pixel sampling of the characters generated by Turbo Pascal V5.5² for images. There are 631 total characters in 5 fonts, distributed as in Table 6.28. Each character was drawn on a VGA monitor with the maximum font size that would allow all characters of a given font to fit within a 16×16 pixel

²Borland International.

	Font 0	Font 1	Font 2	Font 3	Font 4	All Fonts
Number of Characters	253	94	94	95	95	631
Average DFC	44.0	111.3	78.6	96.0	141.9	81.7
Maximum DFC	64	224	160	256	256	256
Minimum DFC	0	24	20	12	24	0
Number with DFC=256	0	0	0	1	11	12

Table 6.29: Character Images DFC Statistics

square. The image was then read and converted into the Ada Function Decomposition (AFD) input format. The results are listed in Table 6.29.

For comparison purposes, the images are listed in Figures 6.10 through 6.14 with their character number and their DFC. The images start with character number 0 in the upper left hand corner and then go across 16 to a row. The rows and columns are numbered such that the number of a character can be found by summing its row and column numbers. The numbers just to the right of each character is the character's DFC. Characters 1-30 of font 3 and characters 21-30 of font 4 are the same as character 31 in each of the fonts. Although these characters are printed in Figures 6.10 through 6.14, they were not run and are not included in the statistics of Table 6.29. There is generally a one-to-one correspondence between dots in the images and ones in the output of the functions that were decomposed. However, there are cases where some differences exist.

In summary, the average DFC for all the characters is 81.7 as compared to 256 for a random function. Only 12 of the 631 characters did not decompose (less than 2 percent). Font 0 was the most patterned and font 4 the least. These results support the contention that DFC measures the pattern-ness of images.

6.3.6 Data as Functions

Data compression depends upon finding some pattern in the data, for example, that there are long strings of blanks or that characters are repeated many times. A central thesis of PT 1 has been that function decomposition is a way to recognize almost any kind of pattern. As another test of this thesis we used the AFD program to decompose some files. The results of the decomposition are then compared to the compression achieved by two popular programs for the PC, PKZIP and PKARC³.

AFD is limited to running on functions of about 9 variables or 512 points. This limits direct application of AFD to files of 64 bytes. We ran AFD version 4a on five files:

- fgfile.pas — the first 64 characters of a pascal program.

³PKWare Inc. Glendale, WI. 1987.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Ⓐ ₄₀	Ⓑ ₄₀	Ⓒ ₄₄	Ⓓ ₆₄	Ⓔ ₄₈	Ⓕ ₄₀	Ⓖ ₁₂	Ⓢ ₁₂	Ⓣ ₄₄	Ⓤ ₄₄	Ⓦ ₄₄	Ⓧ ₄₀	Ⓨ ₄₄	Ⓩ ₃₆	ⓐ ₄₄	ⓑ ₃₆
16	Ⓕ ₄₈	Ⓖ ₆₄	Ⓙ ₄₀	!! ₁₆	Ⓢ ₆₄	Ⓔ ₄₈	≡ ₂₀	Ⓙ ₄₀	↑ ₄₀	↓ ₄₀	→ ₄₄	← ₄₈	Ⓙ ₃₆	Ⓢ ₄₈	Ⓕ ₄₈	Ⓢ ₆₄
32	! ₁₆	" ₁₂	## ₄₈	\$ ₆₄	% ₆₄	& ₆₄	' ₂₄	(₄₈)) ₄₈	* ₄₈	+ ₄₄	, ₃₂	- ₂₀	. ₂₀	/ ₆₄	
48	0 ₃₆	1 ₄₁	2 ₆₄	3 ₆₄	4 ₆₄	5 ₆₄	6 ₆₄	7 ₆₄	8 ₃₆	9 ₆₄	: ₁₆	; ₄₄	< ₄₄	= ₂₀	> ₄₄	? ₆₄
64	Ⓐ ₆₄	Ⓑ ₆₄	Ⓒ ₃₆	Ⓓ ₄₈	Ⓔ ₄₄	Ⓕ ₆₄	Ⓖ ₆₄	Ⓙ ₃₆	Ⓢ ₆₄	Ⓣ ₃₆	Ⓤ ₆₄	Ⓦ ₄₈	Ⓧ ₃₆	Ⓨ ₄₄	Ⓩ ₆₄	ⓐ ₃₆
80	ⓐ ₄₀	ⓑ ₄₈	Ⓒ ₃₆	Ⓓ ₆₄	Ⓔ ₃₂	Ⓕ ₃₆	Ⓖ ₄₀	Ⓙ ₄₄	Ⓢ ₄₄	Ⓣ ₄₀	Ⓤ ₆₄	Ⓦ ₃₂	Ⓧ ₄₄	Ⓨ ₃₂	Ⓩ ₃₂	_ ₈
96	' ₂₄	a ₆₄	b ₆₄	c ₆₄	d ₆₄	e ₆₄	f ₄₄	g ₆₄	h ₄₈	i ₄₄	j ₄₈	k ₄₈	l ₄₄	m ₆₄	n ₄₄	o ₄₄
112	ⓐ ₆₄	ⓑ ₆₄	Ⓒ ₄₈	Ⓓ ₆₄	Ⓔ ₄₀	Ⓕ ₄₄	Ⓖ ₄₈	Ⓙ ₄₈	Ⓢ ₆₄	Ⓣ ₆₄	Ⓤ ₆₄	Ⓦ ₆₄	Ⓧ ₄₄	Ⓨ ₁₂	Ⓩ ₄₄	ⓐ ₆₄
128	Ⓕ ₆₄	Ⓖ ₄₄	Ⓙ ₆₄	Ⓢ ₆₄	Ⓣ ₆₄	Ⓤ ₆₄	Ⓦ ₆₄	Ⓧ ₆₄	Ⓨ ₆₄	Ⓩ ₆₄	ⓐ ₆₄	ⓑ ₆₄	Ⓒ ₆₄	Ⓓ ₆₄	Ⓔ ₆₄	Ⓕ ₆₄
144	Ⓔ ₄₈	Ⓕ ₆₄	Ⓖ ₆₄	Ⓙ ₃₆	Ⓢ ₄₈	Ⓣ ₄₈	Ⓤ ₆₄	Ⓦ ₆₄	Ⓧ ₆₄	Ⓨ ₄₈	Ⓩ ₃₆	ⓐ ₄₀	ⓑ ₆₄	Ⓒ ₄₀	Ⓓ ₆₄	Ⓔ ₆₄
160	Ⓕ ₆₄	Ⓖ ₄₈	Ⓙ ₄₈	Ⓢ ₆₄	Ⓣ ₆₄	Ⓤ ₆₄	Ⓦ ₆₄	Ⓧ ₆₄	Ⓨ ₆₄	Ⓩ ₆₄	ⓐ ₆₄	ⓑ ₆₄	Ⓒ ₆₄	Ⓓ ₆₄	Ⓔ ₆₄	Ⓕ ₆₄
176	ⓐ ₄₈	ⓑ ₄₈	Ⓒ ₄₈	Ⓓ ₄₈	Ⓔ ₄₈	Ⓕ ₄₈	Ⓖ ₄₈	Ⓙ ₄₈	Ⓢ ₄₈	Ⓣ ₄₈	Ⓤ ₄₈	Ⓦ ₄₈	Ⓧ ₄₈	Ⓨ ₄₈	Ⓩ ₄₈	ⓐ ₄₈
192	Ⓕ ₃₆	Ⓖ ₂₀	Ⓙ ₂₀	Ⓢ ₂₄	Ⓣ ₂₀	Ⓤ ₂₄	Ⓦ ₂₀	Ⓧ ₂₄	Ⓨ ₂₀	Ⓩ ₂₄	ⓐ ₂₄	ⓑ ₂₀	Ⓒ ₂₄	Ⓓ ₂₀	Ⓔ ₂₄	Ⓕ ₂₄
208	ⓐ ₂₀	ⓑ ₂₄	Ⓒ ₂₀	Ⓓ ₂₄	Ⓔ ₂₀	Ⓕ ₂₄	Ⓖ ₂₀	Ⓙ ₂₄	Ⓢ ₂₀	Ⓣ ₂₄	Ⓤ ₂₀	Ⓦ ₂₄	Ⓧ ₂₀	Ⓨ ₂₄	Ⓩ ₂₀	ⓐ ₂₄
224	Ⓕ ₄₈	Ⓖ ₆₄	Ⓙ ₆₄	Ⓢ ₃₆	Ⓣ ₆₄	Ⓤ ₆₄	Ⓦ ₆₄	Ⓧ ₆₄	Ⓨ ₆₄	Ⓩ ₆₄	ⓐ ₆₄	ⓑ ₆₄	Ⓒ ₆₄	Ⓓ ₆₄	Ⓔ ₆₄	Ⓕ ₆₄
240	≡ ₂₀	± ₂₈	≥ ₆₄	≤ ₆₄	Ⓙ ₃₆	Ⓢ ₄₀	Ⓣ ₃₂	Ⓤ ₄₀	Ⓦ ₂₄	Ⓧ ₂₀	Ⓨ ₂₀	Ⓩ ₂₈	ⓐ ₂₄	ⓑ ₃₆	Ⓒ ₁₆	

Figure 6.10: Font 0 Images and DFC

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	! ₂₄	" ₄₄	## ₁₀₈	\$ ₁₆₄	% ₂₂₄	& ₂₂₄	' ₃₂	(₅₂)) ₉₆	* ₈₄	+ ₃₂	, ₃₂	- ₂₄	. ₂₄	/ ₉₂	
48	0 ₁₀₄	1 ₆₀	2 ₁₆₀	3 ₁₅₆	4 ₁₆₄	5 ₁₈₄	6 ₁₅₆	7 ₁₅₆	8 ₁₀₈	9 ₁₅₆	: ₂₈	; ₄₀	< ₁₀₈	= ₂₈	> ₉₈	? ₁₁₂
64	Ⓐ ₂₂₄	Ⓑ ₁₈₄	Ⓒ ₁₁₂	Ⓓ ₁₀₈	Ⓔ ₁₃₂	Ⓕ ₁₃₆	Ⓖ ₁₅₂	Ⓙ ₉₄	Ⓢ ₁₃₆	Ⓣ ₉₂	Ⓤ ₁₈₀	Ⓦ ₁₈₈	Ⓧ ₁₈₈	Ⓨ ₁₈₆	Ⓩ ₁₁₂	
80	ⓐ ₁₁₂	ⓑ ₂₂₄	Ⓒ ₁₇₆	Ⓓ ₁₁₂	Ⓔ ₇₆	Ⓕ ₁₁₆	Ⓖ ₁₅₆	Ⓙ ₁₉₆	Ⓢ ₁₉₂	Ⓣ ₁₆₈	Ⓤ ₄₀	Ⓦ ₉₂	Ⓧ ₃₆	Ⓨ ₈	Ⓩ ₋₂₄	
96	' ₂₈	a ₁₂₄	b ₁₁₆	c ₉₆	d ₁₁₆	e ₉₆	f ₉₆	g ₁₇₂	h ₁₁₂	i ₄₀	j ₆₄	k ₁₈₄	l ₄₀	m ₁₂₀	n ₁₁₂	o ₁₂₄
112	ⓐ ₁₁₆	ⓑ ₈₄	Ⓒ ₇₆	Ⓓ ₁₂₀	Ⓔ ₁₀₄	Ⓕ ₁₁₆	Ⓖ ₁₄₀	Ⓙ ₁₇₆	Ⓢ ₁₇₂	Ⓣ ₁₇₆	Ⓤ ₁₂₀	Ⓦ ₇₆	Ⓧ ₂₈	Ⓨ ₇₂	Ⓩ ₅₆	

Figure 6.11: Font 1 Images and DFC

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!28	"24	#56	\$112	%112	&108	'36	(94)80	*112	+44	,40	-28	•32	/68
48	Ü80	190	2132	3104	4124	5152	6160	7120	880	9112	:40	;52	<88	=28	>88	?96
64	Ê140	Ä110	B90	C92	D80	E64	F60	G108	H48	I52	J112	K176	L44	M112	N112	O80
80	P84	Q112	R108	S100	T52	U64	V88	W88	X92	Y92	Z128	[48	\60]48	^64	—
96	'52	ä88	b90	c56	d76	e92	f88	q56	h64	i56	j28	k88	l56	m60	n56	o60
112	P72	q56	r94	s90	t68	u60	v88	w60	x88	y52	z100	{96	20	}92	~68	ü100

Figure 6.12: Font 2 Images and DFC

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ
16	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ	Ⓢ256
32		!24	"44	#108	\$156	%224	&208	'32	(80)72	*84	+52	,32	-24	•24	/92
48	Q120	!52	2164	3152	4140	5160	6120	776	8176	9140	:28	:40	<108	=28	>88	?104
64	Ⓢ224	Ä160	B76	C116	D108	E52	F48	G152	H56	I28	J84	K72	L28	M140	N148	O108
80	P84	Q160	R116	S156	T32	U80	V124	W160	X128	Y104	Z148	[12	\88]12	^52	-28
96	'32	Ⓢ112	b120	c112	d120	e120	f68	g120	h116	i28	j24	k140	l28	m116	n80	o80
112	P120	q120	r76	s120	t48	u60	v92	w112	x112	y116	z104	{76	12	}72	~108	

Figure 6.13: Font 3 Images and DFC

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16						†	†	†	†	†	†	†	†	†	†	†84
32		!64	"44	#108	\$164	%224	&256	'32	(52)96	*84	+52	,32	-24	•36	/92
48	Ü12	190	2164	3140	4152	5132	6224	7116	8176	9208	:44	;56	<108	=28	>88	?132
64	Ⓢ224	Ä256	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ224	Ⓢ256
80	P208	Q256	Ⓢ256	Ⓢ224	Ⓢ208	Ⓢ256	Ⓢ200	Ⓢ176	X256	Ⓢ224	Z184	[40	\92]36	^80	-28
96	'22	Ⓢ126	k160	r112	d140	f136	f104	g156	h104	i64	j76	k140	l76	m172	n132	o132
112	P156	q124	r116	s156	t68	u140	u96	w160	x172	y156	z136	{76	†96	†72	~40	

Figure 6.14: Font 4 Images and DFC

- `story.in` — a 64 character sentence.
- `command.com` — the first 64 characters of an executable file.
- `xor7.cht` — the first 64 characters of a graphics file.
- `parity8.fng` — the first 64 characters of a numeric data file.

Only `parity8.fng` decomposed, to a cost of 220. The size of all these files increased, typically to around 90 bytes when PKARC'ed. Apparently, there is no interesting compression activity for files that are this small.

In order to allow us to run larger files, we considered each bit of a byte to be a different function. Now we can look at 512 byte files as 8 functions each on 9 variables. We ran several files like this.

- `cin.asc`, 537 bytes, a text file containing the first 512 characters of Chapter 1.
- `hawaii.asc`, 524 bytes, a text file containing the first paragraph of an example data text file from enableOA⁴.
- `data.asc`, 519 bytes, a section of the numerical data from the FERD experiment.
- `dbf.asc`, 520 bytes, a section of the example data base file from enableOA.
- `west.muz`, 519 bytes, a section of a file of music data for Pianoman⁵.
- `auld.muz`, 519 bytes, a section of a file of music data for Pianoman.
- `mousea.ss`, 516 bytes, a section from the first part of VGA image of a mouse.
- `mouseb.ss`, 516 bytes, a section from the middle of VGA image of a mouse.
- `mousec.ss`, 516 bytes, a section from the last part of VGA image of a mouse.

The results for each run are listed in Table 6.30 where the rows identified as 1, 2, etc. are the DFC for output bits 1, 2, etc. of the file labeling the appropriate column. The row labeled "Total" is the sum of the DFC's for all eight bits. The rows labeled PKARC and PKZIP are the compressed file size in bits. These results are summarized in Table 6.31, where L is the length of an encoding of the decomposition.

We also generated some functions with trivial decompositions to see what PKARC and PKZIP would do with them.

- (file name: `rand01`) A random string of ASCII 0's and 1's. The file had 514 bytes. Bits 1-7 had zero DFC and bit 8 did not decompose. Therefore, DFC = 512 or 12.5 percent, L = 646 or 15.8 percent, PKARC = 173 bytes or 33.7 percent, PKZIP = 244 bytes or 47.4 percent.

⁴Trademark of Enable Software Inc.

⁵Neil J. Rubenking, 1988.

Bit	cin	hawaii	data	dbf	west	auld	mousea	mousec	mouseb
1	0	0	0	0	36	92	236	512	512
2	512	512	0	512	42	164	512	512	512
3	148	180	0	228	92	104	368	512	512
4	512	512	512	512	76	116	512	320	512
5	512	512	316	512	0	0	348	512	512
6	512	512	276	120	68	124	512	512	512
7	512	512	416	288	40	80	512	512	512
8	512	512	512	320	96	132	512	512	512
Total	3220	3252	2032	2492	450	812	3512	3904	4096
PKARC	3232	2336	1688	2336	1672	1664	2664	3056	2920
PKZIP	3872	2968	2344	2968	1688	1784	3328	3728	3592

Table 6.30: DFC and Data Compression Results for Typical Files

Typical Files:	ARC%	DFC%	L%	ZIP%
auld	40.1	19.8	88.1	43.0
west	40.3	11.0	88.1	40.7
data	40.7	49.6	64.0	56.5
dbf	51.2	60.8	88.1	71.3
mousea	64.5	85.7	100.0	80.6
mouseb	70.7	100.0	100.0	87.0
mousec	74.0	95.3	100.0	90.3
cin	75.2	78.6	87.7	91.1
hawaii	80.7	79.4	88.1	96.6
Average:	56.4	64.5	82.0	70.2

Table 6.31: Data Compression Summary for Typical Files

Contrived files:	ARC%:	DFC%:	L%:	ZIP%:
zeros	8.0	0.0	3.7	30.5
spaces	8.7	6.8	27.8	47.3
zeroone	16.7	0.0	3.7	33.7
majgate	26.6	2.3	15.8	45.1
rand01	33.7	12.5	15.8	47.4
asciistr	94.4	0.0	3.7	85.5
Average:	31.4	3.6	11.8	48.3
All files Average:	48.4	40.1	58.3	63.1

Table 6.32: Data Compression Summary for Atypical Files

- (file name: majgate) A string of ASCII 0's and 1's defining the majority gate function. The file had 514 bytes. Bits 1-7 had zero DFC and bit 8 decomposed to 96. Therefore, DFC = 96 or 2.3 percent, L = 646 or 15.8 percent, PKARC = 136 bytes or 26.6 percent, PKZIP = 232 bytes or 45.1 percent.
- (file name: zeros) A string of ASCII 0's. The file had 514 bytes. All 8 bits had zero DFC. Therefore, DFC = 0 or 0 percent, L = 152 or 3.7 percent, PKARC = 41 bytes or 8.0 percent, PKZIP = 157 bytes or 30.5 percent.
- (file name: zeroone) A string of alternating ASCII 0's and 1's. The file had 516 bytes. All 8 bits had zero DFC. Therefore, DFC = 0 or 0 percent, L = 152 or 3.7 percent, PKARC = 86 bytes or 16.7 percent, PKZIP = 174 bytes or 33.7 percent.
- (file name: spaces) A file with two lines, each line has two ASCII 1's separated by 254 spaces. The file had 516 bytes. Six output bits had zero DFC, the other two had 4 minority elements so their cost is no more than 140. Therefore, DFC = 280 or 6.8 percent, L = 1140 or 27.8 percent, PKARC = 45 bytes or 8.7 percent, PKZIP = 244 bytes or 47.3 percent.
- (file name: asciistr) A file with two lines, each line lists all ASCII characters in order. The file had 517 bytes. All output bits had zero DFC. Therefore, DFC = 0 or 0 percent, L = 152 or 3.7 percent, PKARC = 488 bytes or 94.4 percent, PKZIP = 442 bytes or 85.5 percent.

These results are summarized in Table 6.32.

The ARC, ZIP and AFD data are not directly comparable for several reasons. The AFD representation is "random access" while the ARC and ZIP files must be decompressed as a complete file. The ARC and ZIP compression routines run many times faster than AFD. DFC does not measure a complete representation (i.e. does not measure the interconnection complexity) and AFD does not optimize L . Also, for $n = 9$, as is the case for all these runs, L is bigger than 2^n unless the function has

vacuous variables. That is, L includes a $n^2[A]$ term and for no vacuous variables $[A] > n$. Therefore, $L > n^3 > 2^n$ for $n = 9$. Therefore, we cannot expect decomposition to be a realistic data compression approach for small n . However, despite that, the data compression performance for AFD (as measured by L) is in the same ball-park as ARC and ZIP. That is, ARC had an average compression factor of 0.48 on all files, ZIP 0.63 and AFD 0.58. It is also interesting that there was a high degree of correlation between pattern-ness as measured by DFC and the degree of compression achieved by ARC and ZIP. For the typical files, the correlation coefficient between DFC and ARC was 0.87 and between DFC and ZIP it was 0.94. Remember that the ARC and ZIP compression programs were written by someone who had studied the common kinds of files and had recognized some patterns in these files that allow them to be compressed. ARC and ZIP therefore look for specific kinds of patterns and those kinds of patterns were originally discovered by a person. AFD, on the other hand, finds the patterns itself. The patterns it found in the files allowed compression that was comparable to that of the hand-crafted methods of ARC and ZIP. That there is little pattern-ness beyond that considered by ARC and ZIP is evident in the high degree of correlation between DFC and the ARC and ZIP compression factors for typical files. That there exist some kinds of patterns that ARC and ZIP do not look for is evident in the file `assciistr` where the DFC is 0 yet ARC and ZIP had compression factors between 0.85 and 0.95.

In summary, files were treated as functions and run on AFD. Not only does AFD find patterns in yet another kind of function, it does so as well as programs hand-crafted for this class of patterns. The generality of AFD (and the lack of generality in the hand-crafted programs) is indicated by an example where the compression factor for AFD is less than 5 percent of that of the hand-crafted pattern finders.

6.3.7 Summary

It is our contention that DFC measures the essential pattern-ness that is important in computing. We have the AFD algorithm that estimates DFC (never underestimating though). We took this algorithm and estimated the DFC of as many kinds of non-random functions as we could imagine. This involved about 850 decompositions. Table 6.33 is a summary of all the decompositions of the non-random functions. The DFC column is an average when multiple runs are involved. Table 6.34 shows that larger n tends to have greater decomposability. With the current AFD algorithm, we are just able to decompose functions with sufficiently large n to have interesting patterns; which makes these results all the more remarkable.

Recall that random functions do not decompose with high probability. In all 850 runs there were only about 20 functions that did not decompose. As you look over Table 6.33 notice the correlation between DFC and the intuitive complexity of each function. We believe that this is the most general quantitative measure of pattern-ness ever proposed.

Function	n	m	$m2^n$	DFC	% DFC	Number of runs
XOR	7	1	128	24	18.8	1
count four ones	7	1	128	64	50.0	1
binomial coeff.	8	1	256	20	7.8	1
Lucas function	8	1	256	28	10.9	1
greater than	8	1	256	28	10.9	1
palindrome	8	1	256	28	10.9	1
images of chars	8	1	256	81.7	31.9	631
logarithm	8	3	768	56	7.3	3
square root	8	4	1024	132	12.9	4
GCD	8	4	1024	388	37.9	4
modulus	8	4	1024	280	27.3	4
remainder	8	4	1024	378	36.9	4
addition	8	5	1280	92	7.2	5
subtraction	8	5	1280	218	17.0	5
1 bit sorting	8	8	2048	488	21.9	8
2 bit sorting	8	8	2048	512	25.0	8
multiplication	8	8	2048	892	43.6	8
sine	8	8	2048	1350	65.9	8
parity	9	1	512	32	6.3	1
language accept	9	1	512	46.6	9.1	14
majority gate	9	1	512	96	18.8	1
Fibonacci test	9	1	512	144	28.1	1
cube root	9	3	1536	168	10.9	3
determinant	9	3	1536	270	17.6	3
files	9	8	4096	1644.1	40.1	120
primality test	10	1	1024	600	58.6	1
k-clique	13	1	8192	380	4.6	5

Table 6.33: Decomposition Summary for Non-Randomly Generated Functions

Function	n	m	$m2^n$	DFC	% DFC	Number of runs
multiplication	4	4	64	40	62.5	4
multiplication	6	6	384	164	42.7	6
multiplication	8	8	2048	892	43.6	8
primality test	6	1	64	64	100.0	1
primality test	7	1	128	104	81.2	1
primality test	8	1	256	196	76.6	1
primality test	9	1	512	336	65.6	1
primality test	10	1	1024	600	58.6	1
Fibonacci test	5	1	32	24	75.0	1
Fibonacci test	6	1	64	48	75.0	1
Fibonacci test	7	1	128	76	59.4	1
Fibonacci test	8	1	256	108	42.2	1
Fibonacci test	9	1	512	144	28.1	1
binomial coeff.	6	1	64	12	4.7	1
binomial coeff.	8	1	256	20	7.8	1
Lucas function	6	1	64	20	31.2	1
Lucas function	8	1	256	28	10.9	1
majority gate	7	1	128	48	37.5	1
majority gate	9	1	512	96	18.8	1
parity	7	1	128	24	18.8	1
parity	8	1	256	28	10.9	1
parity	9	1	512	32	6.3	1
4 node k-clique	8	1	256	128	50.0	4
5 node k-clique	13	1	8192	380	4.6	5
files	9	1	256	248.8	97.2	5
files	9	8	4096	1589.2	38.8	120

Table 6.34: Larger n Shows Greater Decomposability

6.4 Patterns as Perceived by People

In this section we consider the relationship between complexity as measured by a function decomposition algorithm and complexity as perceived by people when functions are represented as two-dimensional images.

We have hypothesized that the DFC measure captures the essence of pattern-ness in general. DFC has established correlation with conventional measures such as time complexity, program length and circuit complexity. Since people are, in some sense, a computation system, we wonder: does this measure correlate with complexity as perceived by people? The AFD program provides the DFC measure of complexity. To get human assessments of the complexity of these functions, we turned the function into something which can be easily perceived. There are a variety of possible ways of doing this. An image can be created from a function as discussed in Section 6.3.5. A similar method could be used to produce other experiences (e.g. sounds).

We generated a set of test functions, found the DFC complexity using the AFD program, found the complexity as perceived by people and assessed the relationship between these two measures. Thomas Abraham reports the results of this study in [1]. In summary, this experiment found a correlation coefficient of 0.8 between DFC and pattern-ness as ranked by people.

6.4.1 Effect of the Order of Variables on the Pattern-ness of Images

At various points throughout the PT 1 study it has been useful to think of binary functions as black and white images. This technique was used in the extrapolation experiments, in the tests for DFC generality and in the pattern-ness/DFC correlation study just discussed.

In order to get an image from a binary function one must choose which variables specify rows and which specify columns. You must also decide the order among the row variables and among the column variables. This experiment is concerned with the effect of the chosen grouping of variables on the appearance of the images. Table 6.35 lists the images used in this experiment.

We used the Turbo Pascal character set (as in Section 6.3.5) as a source of functions. A program was used to draw a specified function with a variety of variable permutations. Figures 6.15 through 6.18 show a sequence of images resulting from the program. Each figure has the character number, the font number, and the Decomposed Function Cardinality (DFC) listed at the bottom. At the top left of the figure is the image as drawn by the Turbo Pascal procedure `outtextxy`. At top right is the drawing of the function that was generated by "getting pixels" from the left image. The top two images should always be the same. The order of the variables for the top row's images can arbitrarily be labeled as 1, 2, 3, 4 for the columns and 5, 6, 7, 8 for the rows. From left to right, the second row of images are a permutation of these variables as in Table 6.36. Permutation 1 is a swapping of the high and low

Font No.	Char No.	DFC
0	177	4
0	197	20
0	15	36
0	1	40
0	10	44
0	65	64
2	48	80
2	51	104
2	65	120
3	65	160
3	31	256
1	65	184
4	65	256

Table 6.35: Character Images

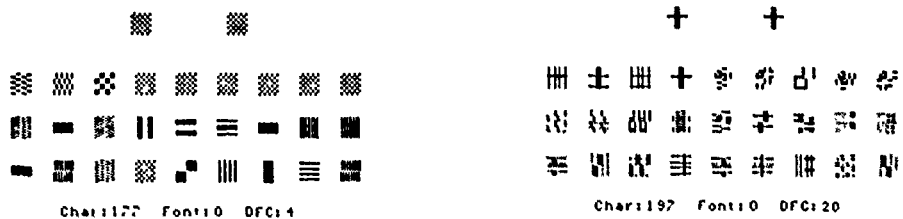


Figure 6.15: Variable Permutations for Characters 177 and 197 of Font 0

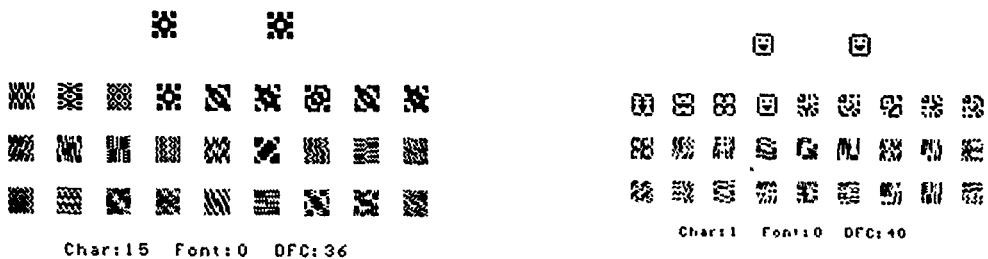


Figure 6.16: Variable Permutations for Characters 15 and 1 of Font 0

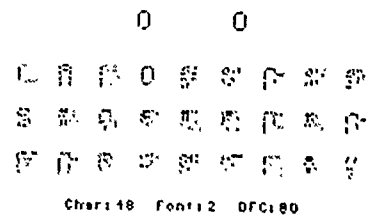
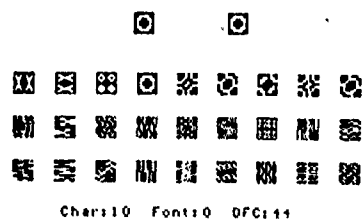


Figure 6.17: Variable Permutations for Characters 10 of Font 0 and 48 of Font 2

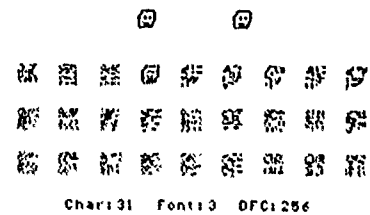
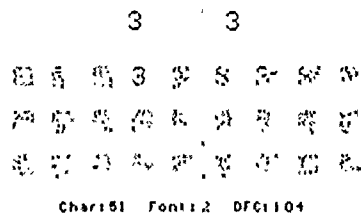


Figure 6.18: Variable Permutations for Characters 51 of Font 2 and 31 of Font 3

Permutation	Column Variables				Row Variables			
Original	1	2	3	4	5	6	7	8
1	4	3	2	1	5	6	7	8
2	1	2	3	4	8	7	6	5
3	4	3	2	1	8	7	6	5
4	5	2	3	4	1	6	7	8
5	1	6	3	4	5	2	7	8
6	1	2	7	4	5	6	3	8
7	1	2	3	8	5	6	7	4
8	1	2	7	8	5	6	3	4
9	1	6	3	8	5	2	7	4

Table 6.36: Permutations of Variables

order bits of the column variables; 2 is a swapping of the high and low order bits of the row variables; 3 is a swapping of the high and low order bits of both the column and row variables; 4 through 7 swap a column variable for a row variable; 8 swaps two side-by-side column variables for two side-by-side row variables; 9 swaps every other column variable for every other row variable. Note that all the font 0 characters have at least two vacuous variables while none of the characters in the other fonts have any vacuous variables.

This data shows that the apparent pattern-ness of a given function is highly dependent upon the variable permutation. For example, character 51 of font 2 with the original combination of variables appears to be highly patterned while all of the images resulting from a random permutations of the variables do not appear to be patterned. This points out the risk in trying to "see" the patterns in a function by turning it into an image. There is at least one definite property, "connected-ness" that the original images have that is lost in the random permutations. All but two of the original images (i.e. all but characters 1 and 177 of font 0) are made up of no more than 3 "islands." The randomly permuted images are made up of many more of these islands. We argue that the "real" abstract pattern-ness of all the images of a particular function is the same and that the difference in apparent pattern-ness is a result of our biological visual system.

6.5 Pattern-ness Relationships for Related Functions

When we defined the PT 1 problem (Section 3.4.9) we chose to only consider functions defined by tables. However, we eventually will consider functions defined in other ways. This raises the question, what does the pattern-ness of one function imply about the pattern-ness of a related function? We are especially interested in this question when one function is used to define a second function. For example, we might be asked to compute $y = f(x)$ where $x = y^2$. What does knowing how to compute x from y tell us about computing y from x ? We hope to study this question in depth in PT 2; however, we have some initial results that are described in this section.

6.5.1 Functions and Their Complements

Perhaps the most trivial kind of relationship is when one function is simply the complement of a second function. If $f(x) = g(\text{not}(x))$ or $\text{not}(g(x))$ or $\text{not}(g(\text{not}(x)))$ then f and g have the same DFC (except when f or g is a projection function). Not only is the DFC the same, so is the architecture of the decomposition (i.e. the algorithm). Therefore, if we are asked to design an algorithm for f and f is defined for us in the form of an algorithm for the complement of f then our job is very simple.

6.5.2 Functions and Their Inverses

This section is concerned with the relationship between the complexity of a function and the complexity of that function's inverse⁶ and summarizes [30].

One motivation for studying this problem comes from the fact that many computational problems are originally specified in terms of the inverse of the function which we want to realize. For example, there is the "classical inverse problem" of computer vision. The transformation from a 3-D model of a space into a 2-D image of that space from a particular perspective (Projective Geometry) is well understood and of relatively low computational complexity. The computer vision problem is easy to specify using Projective Geometry. However, the objective in computer vision is to take a 2-D image and generate from that a 3-D model. Therefore, the problem is specified using a function (Projective Geometry) which is the inverse of the function that we wish to realize on a computer. A second example is the situation where we want to find an x with a particular given property y . Typically, the properties of x (say $P(x)$) are easy to compute and this is how the problem is specified. However, the inverse of P is needed to find x with a given input property y .

The question then arises; if a function (f) is of low computational complexity (which is generally the case when f is used in a problem specification) then should we expect the inverse of f to have low computational complexity. In other words, if a problem has a simple specification then does it necessarily have a simple computer realization?

Our approach to studying this problem was to generate many functions and their inverses and then compute their complexity. Since we are studying only true functions, we required the functions of this study to be bijections. That is, both the function and its inverse are true functions.

This experiment was done with functions of the form:

$$f : \{0, 1\}^4 \rightarrow \{0, 1\}^4.$$

There are actually four functions going each way and the "cost" in the tables are the sum of the DFC's of the four functions. We decomposed about 34,000 function-inverse pairs. Table 6.37 shows the relationship between functions with a given DFC and the average DFC of those function's inverses. Figure 6.19 shows the average DFC relationship between functions and their inverses.

There are function-inverse pairs that have substantially different DFC's, e.g. 36 versus 52 or 40 versus 56. However, they tended to be the same on average. In particular, 19.9 percent of the functions had higher cost than their inverses, 20.2 percent had lower cost and 59.9 percent had the same cost. The experimental correlation coefficient between the DFC of a function and its inverse was 0.90. We observed from the detailed data that if one (or more) of the functions going one way was a projection then the inverse functions would also include the same number of projections. This was also true of complements of projections.

⁶This experiment was performed by John Langenderfer.

	f^{-1} Cost															
f Cost	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62	64
34	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
36	0	0	0	0	0	1	0	0	0	3	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	3	0	1	0	1	0	0	0	0	0	3	0	3
42	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
44	0	2	0	2	0	3	0	6	0	5	0	3	0	5	0	9
46	0	0	1	0	1	0	2	0	2	0	0	0	0	0	0	0
48	0	0	0	2	0	4	0	41	0	20	0	32	0	74	0	60
50	0	0	0	0	1	0	4	0	39	0	0	0	0	0	0	0
52	0	0	0	1	0	9	0	28	0	93	0	124	0	229	0	306
54	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
56	0	0	0	2	0	4	0	25	0	134	0	569	0	1057	0	2039
58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
60	0	0	0	0	0	13	0	57	0	248	0	1115	0	3180	0	5850
62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	0	0	0	0	0	5	0	55	0	309	0	2060	0	5907	0	25710

Table 6.37: Number of Functions and Inverses with a Given Cost Combination

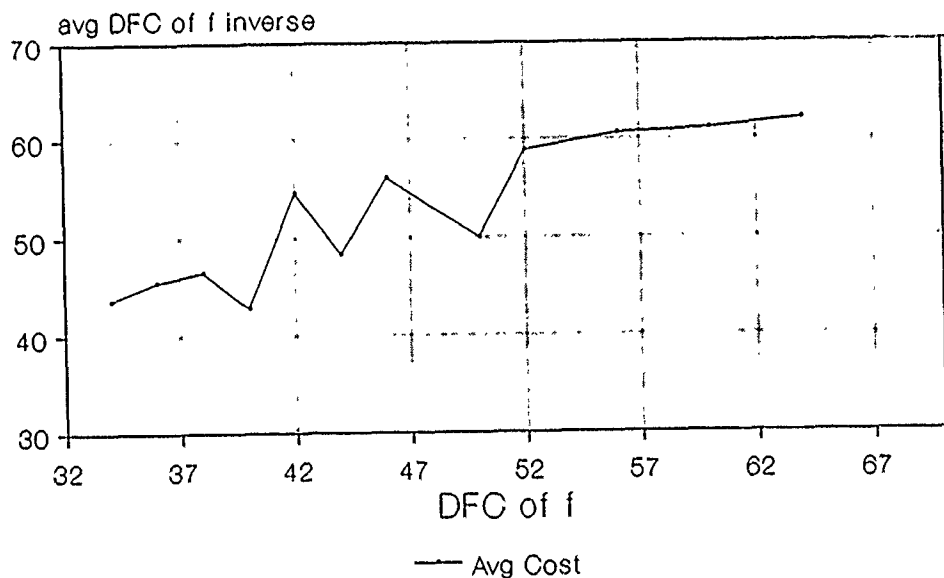


Figure 6.19: Relationship Between Functions of a Given DFC and the Average DFC of Their Inverses

6.6 Extrapolative Properties of Function Decomposition

6.6.1 Introduction

It is possible to divide the algorithm design problem into the problem of defining what function you want the algorithm to compute (the definition problem) and the problem of getting a computer to compute the desired function with the given computing resources (see [50]). For the PT 1 project, we deliberately set aside the definition problem to focus on the realization problem. However, as an aside we did some experiments on one approach to the definition problem. This section describes the results of these experiments. Our Ada Function Decomposition (AFD) program is set up such that it will decompose partial functions. A partial function is a function with some of the outputs undefined or, equivalently, there are inputs for which we don't care what the computer outputs. It turns out that by recomposing a decomposed partial function you end up with a function that is "less partial." Therefore, recomposing decomposed partial functions is an approach to the definition problem. The exploration of this approach will be the main topic in the Pattern Theory 2 project.

How you approach the definition problem depends upon what you are given about the problem. One of the most common forms for giving information about a function is a set of samples. Sometimes you may also be given some information about how to extrapolate these samples but not always. There are a many ways to extrapolate samples (Neural Nets, rule induction, fitting polynomials, nearest-neighbor, etc.); each way is based upon some (often implicit) assumption about the form of the function. How well a given approach works depends upon the validity of that assumption. The traditional approaches to machine learning assume a relatively large amount about the function and therefore have a relatively narrow range of successful applications. Since the range of possible solutions is small, these traditional approaches tend to require only a few samples. One of the most neglected aspects of machine learning research is that of specifying exactly what a particular approach is assuming about the function. Our approach assumes that the desired function has low computational complexity. The central thesis of Pattern Theory (PT) is that a function has structure (i.e. is patterned) if and only if it has low computational complexity. Therefore, the Function Extrapolation by Recomposing Decompositions (FERD) approach to machine learning assumes that the desired function is structured, but does not require a specific kind of structure. We have shown that, while it is highly unlikely that an arbitrary function will be structured, functions of interest in computing tend to be structured. The FERD approach contrasts the traditional approaches where they not only assume that the desired function is structured, but that the function has some specific structure (geometric, syntactic, etc.).

An important principle in science (the Principle of Parsimony or the idea of Occam's Razor) is that one should choose the simplest theory that is consistent with the experimental results. FERD is basically using this same principle in its approach

to machine learning. In regards to human learning, there is much debate about the roles of nature (phylogeny) and nurture (ontogeny). In terms of machine learning, the “nature” may be characterized as the set of assumptions made by the machine learning system and the “nurture” as the sample set (c.f. Figure 2.1). Since FERD assumes nothing about the specific structure of the function to be learned (only that it is structured), knowledge learned by FERD has very little “nature” content. In the trade-off between being able to learn lots of different kinds of things and being able to learn quickly, FERD is unusual in that it sides with diversity.

6.6.2 FERD Experiments

A series of experiments were conducted to assess the performance of FERD. We took a number of functions, sampled them, decomposed the sampled functions, recomposed the decompositions, and compared the recomposed functions to the originals. The “symmetric” function outputs one if and only if the input has exactly four 1’s in it. The other functions are pretty well described by their names.

We varied the number of samples; but, for each number of samples, we always generated five randomly sampled versions of the original function. The sampled versions of the function were then decomposed. The AFD output was then recomposed. The recomposed functions were then compared to the original function. We computed the average, maximum and minimum error for the five sampled versions of the original function for each sample set size. If f is the original function and r the recomposed function then error e_r is defined as the

$$e_r = \sum_{x \in \{0,1\}^n} e(x)$$

where $e(x) = |f(x) - r(x)|$ when $r(x)$ is defined and $e(x) = 1/2$ when $r(x)$ is undefined. The results of these experiments are shown in Figures 6.20 through 6.30.

Each of these graphs has two vertical scales. The left scale is the number of differences between the original function and the recomposed function. Average, minimum and maximum errors are plotted relative to this scale. The recomposed functions may still have some “don’t cares” even after recomposition. The average number of don’t cares in the recomposed function is plotted relative to the left scale as “Avg D-Cares.” The curve labeled “Chance” is the average number of errors that would result from randomly filling in the don’t cares of the original sampled function. One would hope that a machine learning system would do better than chance. The right scale is the Decomposed Function Cardinality (DFC) of the sampled functions. The average (labeled “P-Cost”), minimum and maximum DFC’s are plotted relative to this scale.

Note the cost-error relationship of these curves. There do not seem to be many cases where we can get a large decrease in cost for a small increase in error. If this is true in general then our specialization to zero error realizations (in Chapter 3) may not be as great a loss of generality as we thought.

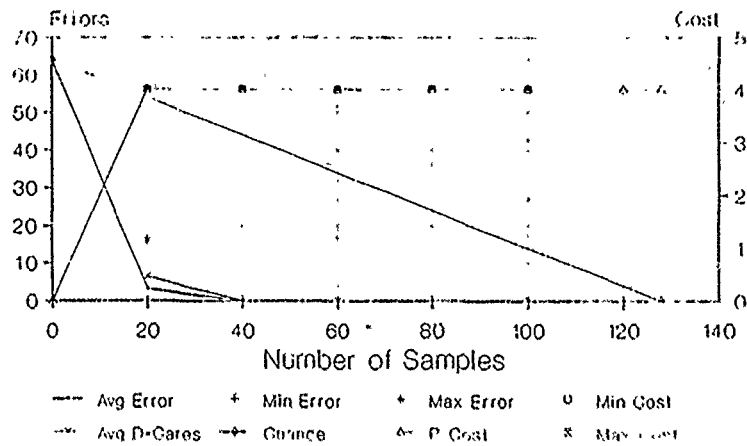


Figure 6.20: Learning Curve for XOR Function

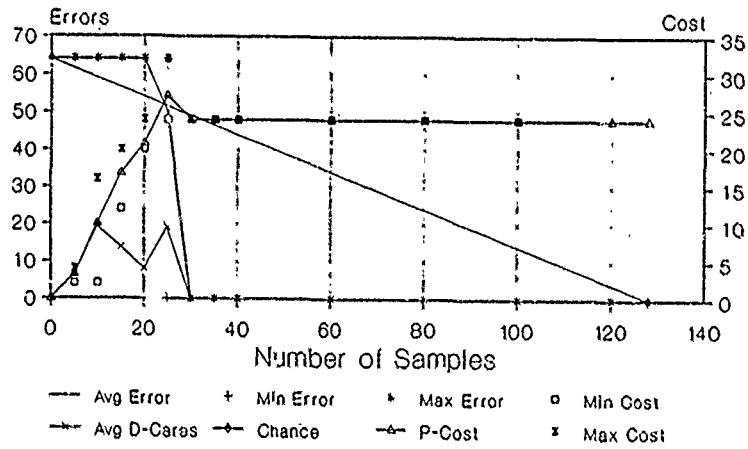


Figure 6.21: Learning Curve for Parity Function

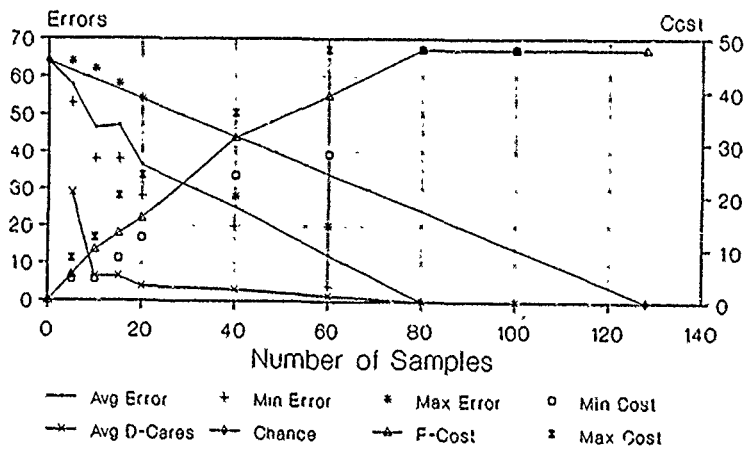


Figure 6.22: Learning Curve for Majority Gate Function

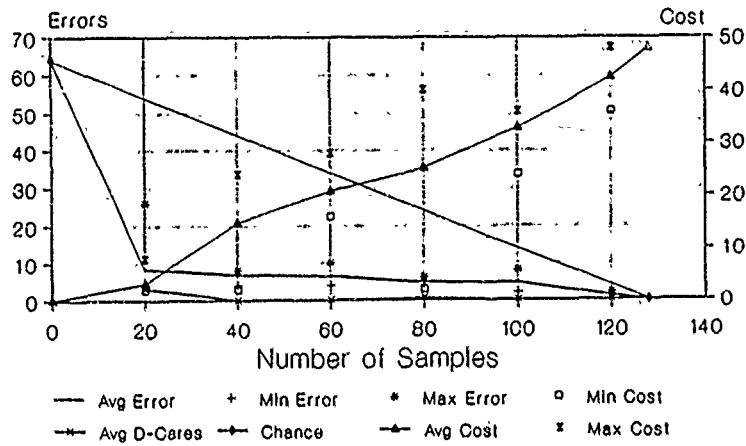


Figure 6.23: Learning Curve for a Random Function with Four Minority Elements

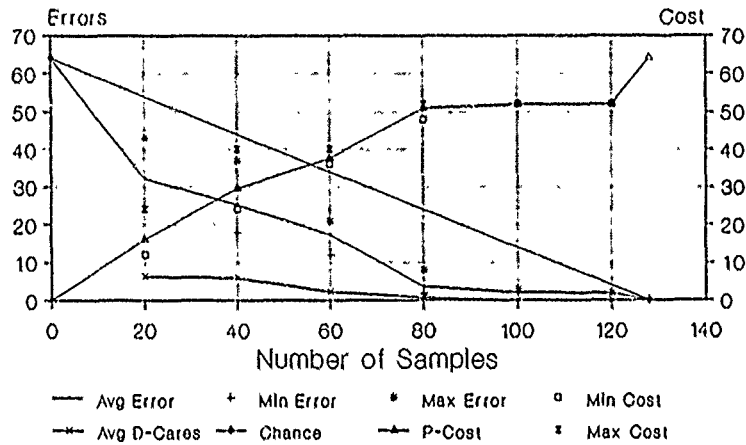


Figure 6.24: Learning Curve for the Symmetric Function

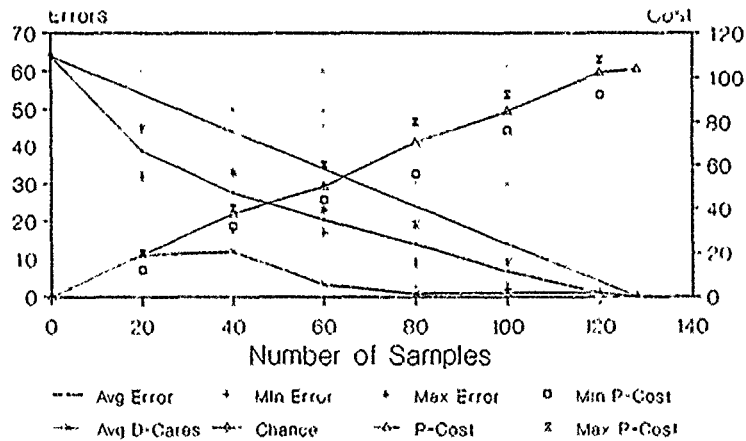


Figure 6.25: Learning Curve for Primality Test on Seven Variables

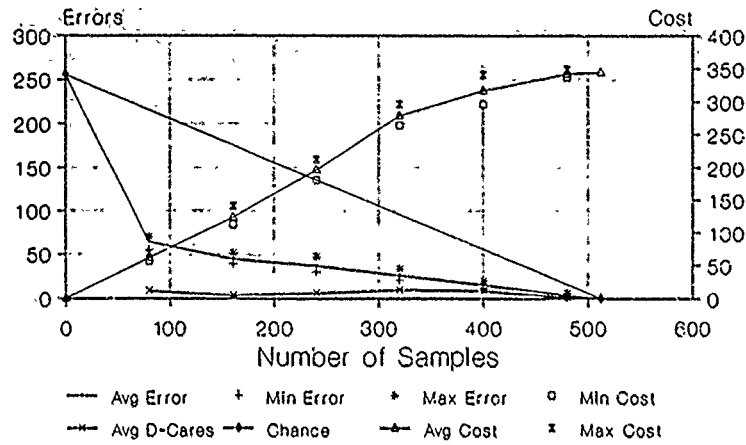


Figure 6.26: Learning Curve for Primality Test on Nine Variables

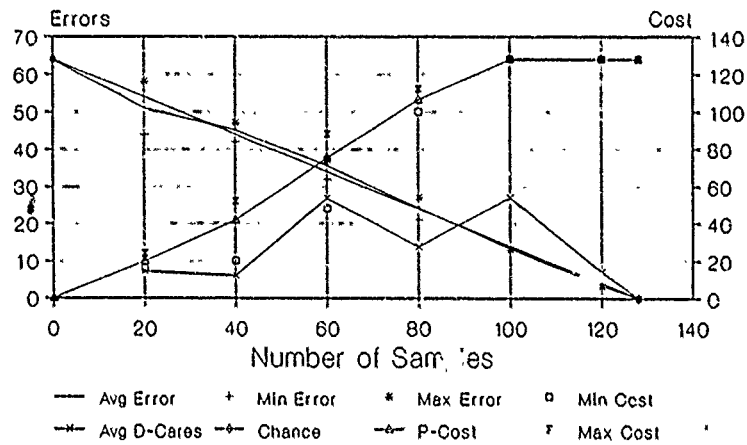


Figure 6.27: Learning Curve for a Random Function

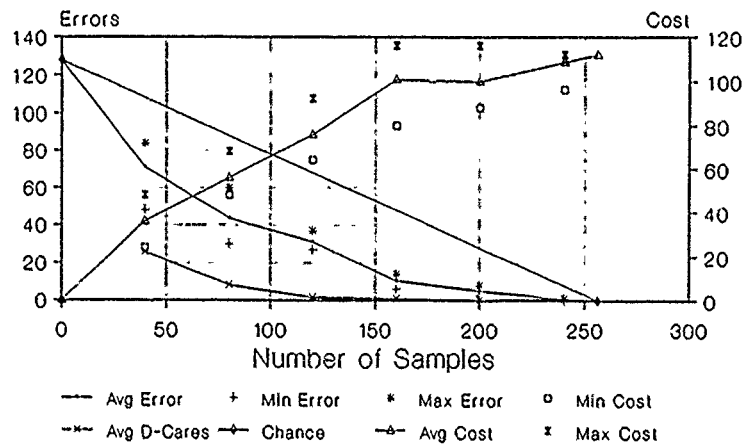


Figure 6.28: Learning Curve for Font 1 "P"

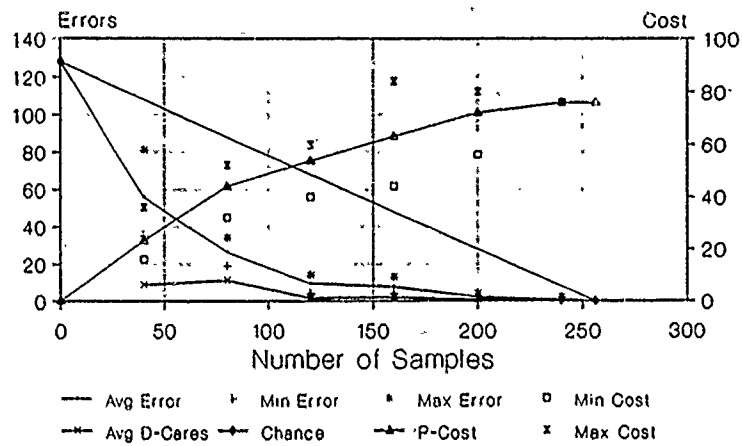


Figure 6.29: Learning Curve for Font 1 "T"

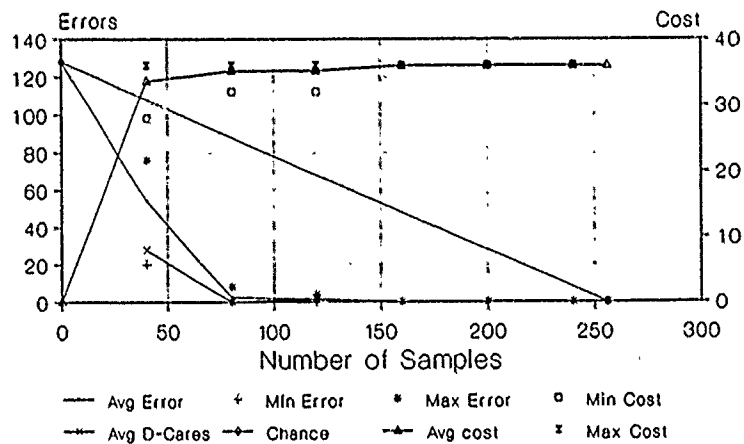


Figure 6.30: Learning Curve for Font 0 "R"

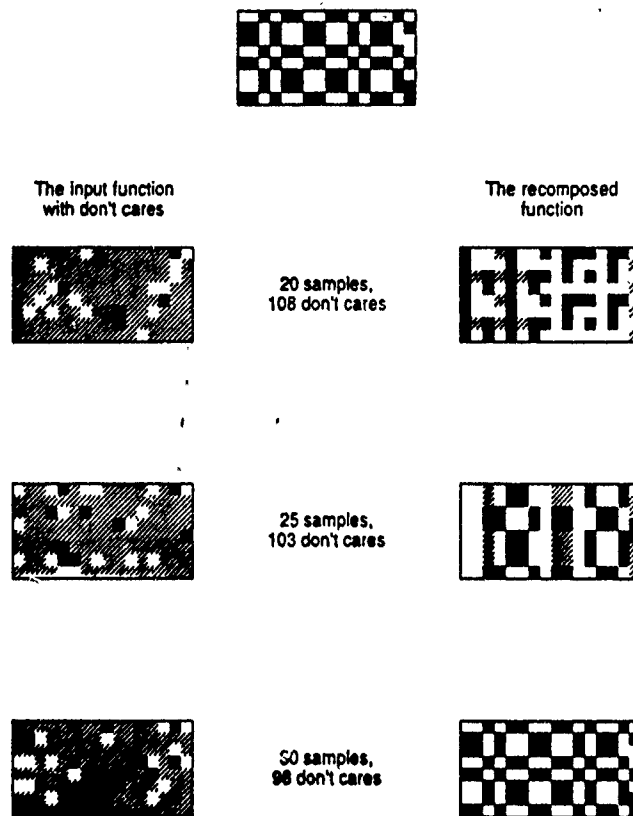
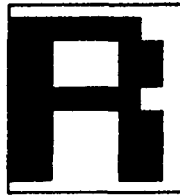


Figure 6.31: Learning Examples for the Parity Function

A sample of the results are also shown in picture form for the parity function and the letter "R" in Figures 6.31 and 6.32. Figure 6.33, showing FERD results for "P" and "T", is the Pattern Theory logo. Figure 6.34 shows the error statistics for a random function on each of 4 through 10 binary variables. Only one sampled version of each original function for each sample set size was made for this graph.

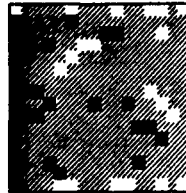
The results of these experiments were very consistent. As the DFC of the original function goes up, the number of samples required for a given error rate goes up. That is, complex functions are harder to learn. This is demonstrated in Figure 6.35, which shows the minimum number of samples required for less than 10 errors as a function of DFC. For random functions, FERD did no better than chance. Also, the error rate tended to go down in rough proportion to the increase in the DFC of the learned function. Functions with a definite minority of minority elements (such as the symmetric function, RND4ONES, and PRIMES 9) tended to have a plateau in their error curves. A random selection of samples would contain few minority elements. We would not expect to see this plateau if we had balanced the samples between function elements with an output of 0 and those with an output of 1.

Diagram 3.2
char. 82

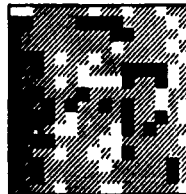
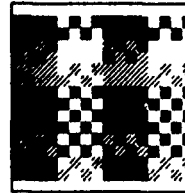


The input function
with don't cares

The recomposed
function



40 samples,
216 don't cares



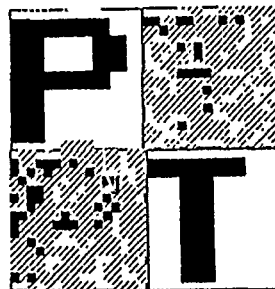
80 samples,
176 don't cares



120 samples,
136 don't cares



Figure 6.32: Learning Examples for the Letter "R "



PATTERN THEORY

Figure 6.33: The Pattern Theory Logo

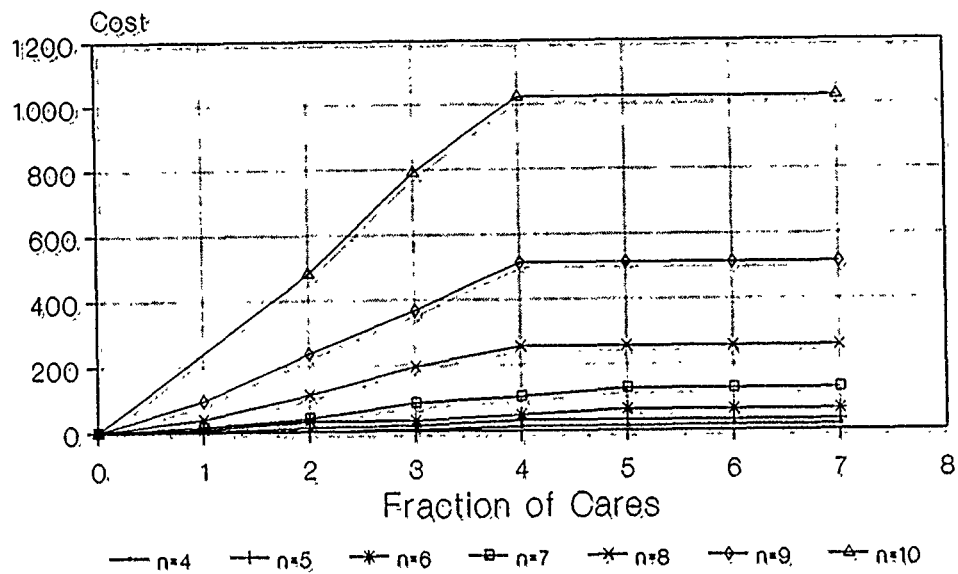


Figure 6.34: Learning Curves for Random Functions on 4 Through 10 Variables

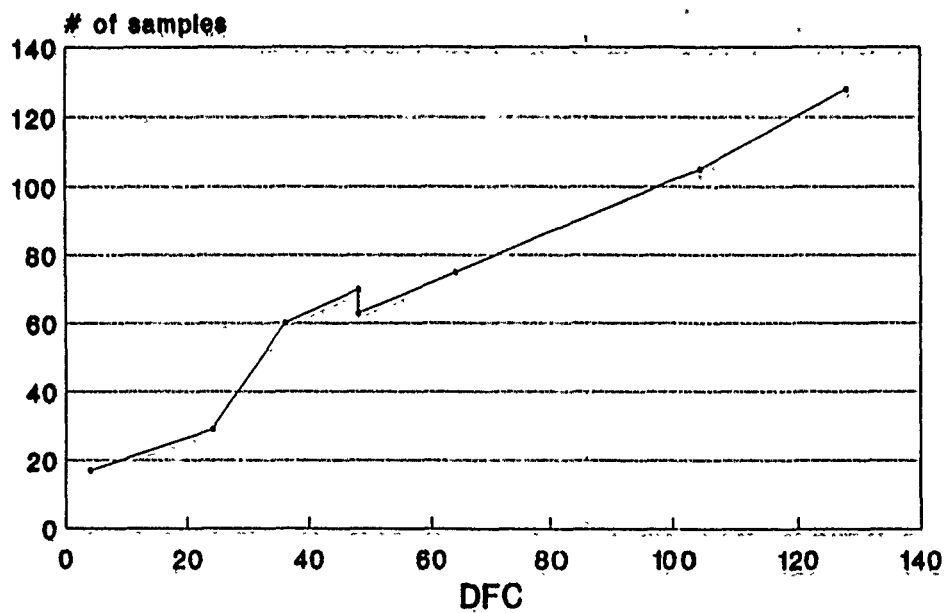


Figure 6.35: Number of Samples Required for ≤ 10 errors

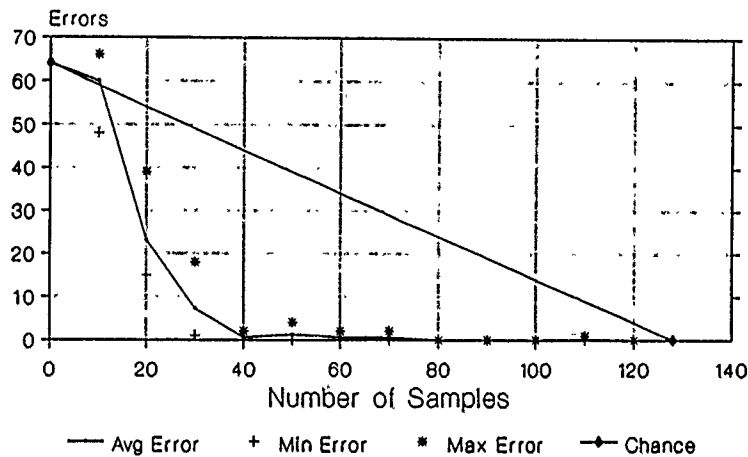


Figure 6.36: Neural Net Learning Curve for XOR Function

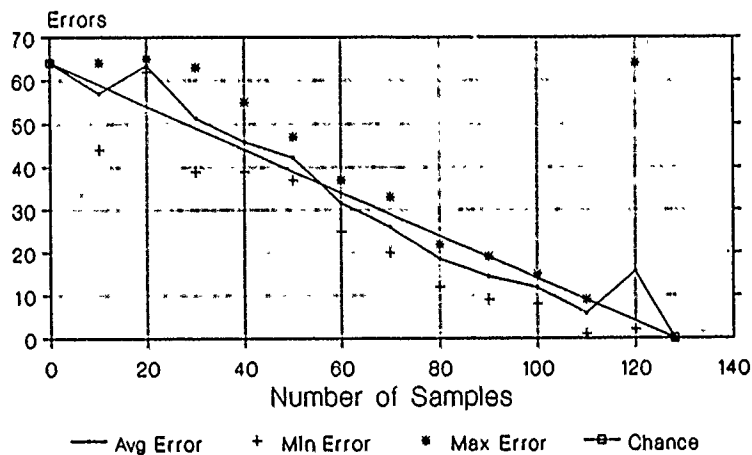


Figure 6.37: Neural Net Learning Curve for Parity Function

6.6.3 FERD and Neural Net Comparisons

We repeated some of the above experiments using a neural net (NN) rather than FERD.⁷ The neural net had three layers with n nodes in the first layer, $2n + 1$ nodes in the second layer and 1 node in the final layer. The weights for the input layer were fixed and the weights for the other layers were trained using back-propagation as defined in [10, pp.53-59]. These results are shown in Figures 6.36 through 6.39. Measuring the performance of a neural net is not as black and white as in the FERD approach. Neural nets have a number of user specified parameters and architectural features that affect their performance. These parameters are used as a way for a human to get more "nature" into the machine to get it started learning. Therefore, it may well be possible to fine tune some neural net to get better performance than we

⁷Thomas Gearhart performed most of these experiments.

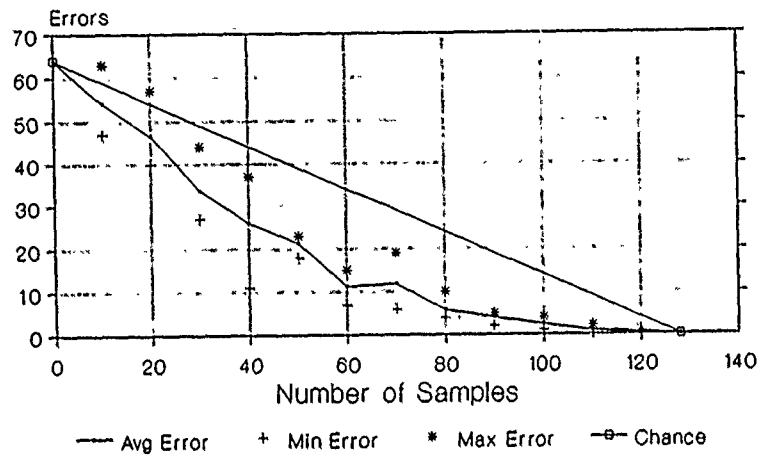


Figure 6.38: Neural Net Learning Curve for Majority Gate Function

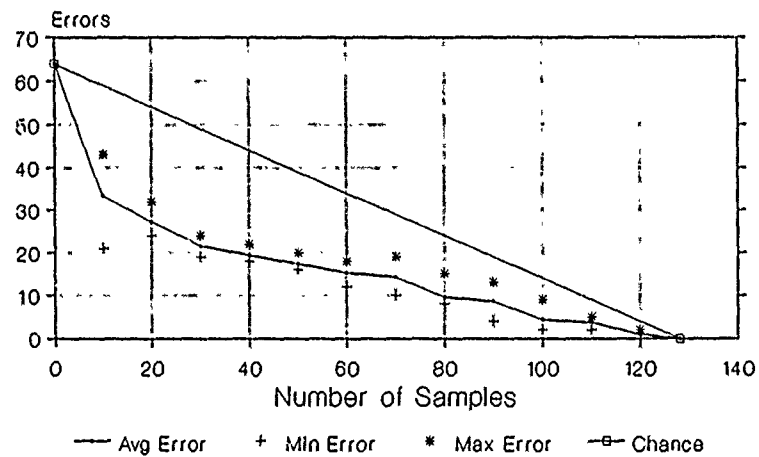


Figure 6.39: Neural Net Learning Curve for the Symmetric Function

	20		40		60		80		100		120		Total		Total %	
Function	F	N	F	N	F	N	F	N	F	N	F	N	F	N	F	N
Symm	32	27	25	19	18	15	4	10	2	4	2	1	83	76	10.8	9.9
MajGate	37	46	26	26	12	11	0	6	0	3	0	0	75	92	9.8	12.0
Parity	64	63	0	46	0	31	0	18	0	11	0	15	64	184	8.3	24.0
XOR	4	23	0	1	0	1	0	0	0	0	0	0	4	25	0.5	3.3
Total	137	159	51	92	30	58	4	34	2	18	2	16	226	337	7.4	11.0

Table 6.38: FERD (F) and NN (N) Error Comparison

got here; but since FERD requires no fine tuning, we felt this was a fair comparison. Table 6.38 compares the errors from the two approaches. For the Symmetric function the neural net did slightly better than FERD (9.9 percent average error versus 10.8 percent); but on all other functions, the Neural Net did worse, sometimes much worse. The lack of generality of the Neural Net approach is seen in its performance on the parity function. Here the Neural Net did no better than chance while FERD learned the function exactly with 30 (of 128 total points) samples. Over all functions and sample set sizes, the NN had an average error rate of 11 percent compared to FERD's 7.4 percent. Note that a random extrapolation of any function would result in a composite average error rate of about 25 percent. FERD got the function exactly right in 13 cases compared to 4 for the NN.

Another limitation of Neural Nets became apparent when we implemented a second NN, this one had n nodes in the first layer, 10 nodes in the second layer, 5 nodes in the third layer and 1 node in the final layer. The back-propagation implementation of this NN was taken from [53]. The learning curves for this NN are shown in Figure 6.40 and Figure 6.41. The points at 0 and 128 samples do not represent actual data. Figure 6.40 demonstrates that this NN worked well for a step function. Notice that although the first NN performed well on the Majority Gate function, this second NN showed no consistency. This points out that selecting an architecture for the net is important. However, this selection is left to the designer with no theory to say what architecture is appropriate for a given class of functions. This contrasts the FERD approach, which "solves" for the architecture as a function of the data.

6.6.4 FERD Theory

We were surprised that the Neural Net did better than chance and even more surprised that FERD did better than a Neural Net. Our best theoretical explanation for this is based on the probability that samples have a certain degree of structure. Reference [33, p.194] identifies Laplace and Kolmogorov as having recognized that regularity consistent with a simple law probably is a result of that simple law.

Let F be the set of all binary functions on n variables. Rather than use the simple DFC, our cost measure is program length as defined in Section 4.3. This cost is es-

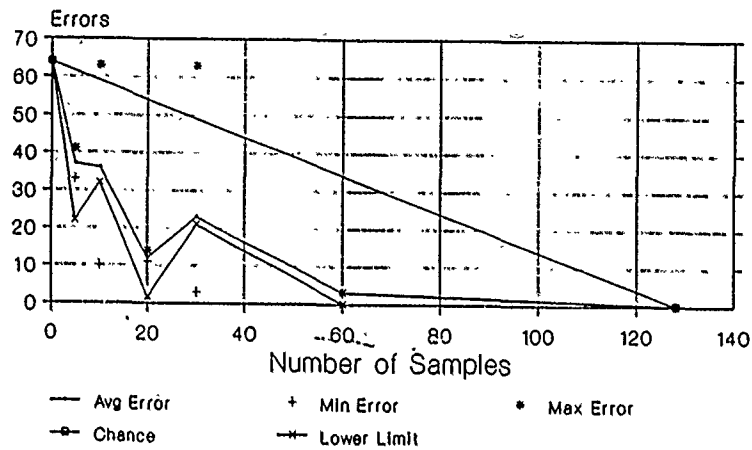


Figure 6.40: Second Neural Net's Learning Curve for the Step Function

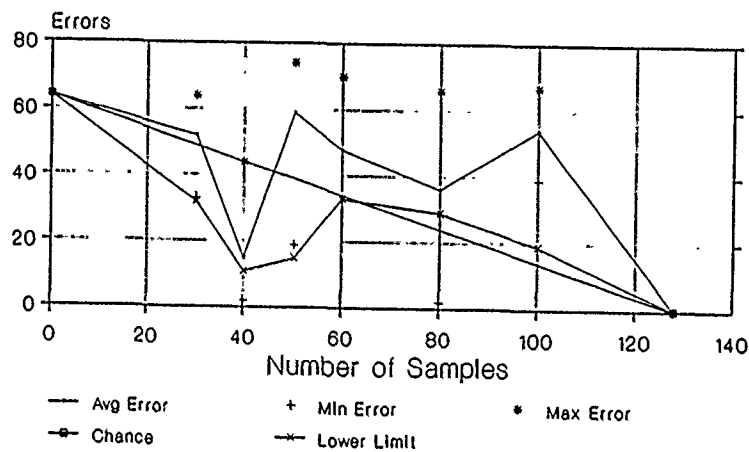


Figure 6.41: Second Neural Net's Learning Curve for the Majority Gate Function

entially DFC with something added to reflect the complexity of the interconnections of the decomposition's components. There is some function (f) that we want to learn and we have a set of samples from f . There are two subsets of F that are of special interest. One subset (S) is the set of all the functions consistent with the samples. The other is the set (C) of all functions with a cost less than or equal to the cost of f . The function f is in both these subsets and FERD is such that it always produces a learned function (g) that is also in both these subsets.⁸ That is, f and g are always consistent with the samples and will have a cost that does not exceed f 's. Therefore, the size of the intersection of these two sets tells us something about how far g can be from f . In particular, when the intersection only has one member, $g = f$ (i.e. FERD gets it exactly right). We use $[A]$ to denote the cardinality of a set A . Let us summarize our notation:

- F : the set of all functions of the form $f : \{0,1\}^n \rightarrow \{0,1\}$, $[F] = 2^{(2^n)}$.
- f : some function in F that we want to learn.
- c : the cost of f .
- s : the number of samples that we are given from f .
- S : the set of functions from F that are consistent with the samples. $[S] = 2^{2^n - s}$.
- C : the set of functions from F that have cost less than or equal to c . $[C] \leq 2^c$ since we can name all the elements in C with c bits (see Theorem A.8).
- g : the function produced by FERD from the samples of f .

The hypergeometric probability distribution [41, pp.175-176] is of interest here. A random variable X has a hypergeometric distribution with parameters N, n and r if

$$P(X = k) = \frac{\binom{r}{k} \binom{N-r}{n-k}}{\binom{N}{n}}, k = 0, 1, 2, \dots$$

Meyers'70 gives the following useful properties of the hypergeometric distribution. Let $p = r/N$ and $q = 1 - p$:

- $E(X) = np$,
- $V(X) = npq(N - n)/(N - 1)$,
- $P(X = k) \simeq \binom{n}{k} p^k (1 - p)^{n-k}$ for large N .

⁸Figures 6.21 and 6.29 have points where the cost of an extrapolated function exceeds the original function. This is a result of the non-optimal nature of the software used and should not happen in principle.

The last property indicates that the hypergeometric distribution is well approximated by the binomial distribution when N is large. A binomial distribution with the above parameters has expected value np and variance npq .

Given two subsets A and B of a universal set U , if A and B are randomly and independently selected from U then the number of common elements has the hypergeometric distribution with parameters $[U]$, $[A]$ and $[B]$; that is, for X the size of $A \cap B$,

$$P(X = k) = \frac{\binom{[A]}{k} \binom{[U] - [A]}{[B] - k}}{\binom{[U]}{[B]}}, k = 0, 1, 2, \dots$$

Now let $p = [A]/[U]$ and $q = 1 - p$:

- $E(X) = [B]p$,
- $V(X) = [B]pq([U] - [B])/([U] - 1)$,

Returning now to our problem and our notation, if C and S are independently and randomly selected subsets of F then the cardinality of C intersect S has a hypergeometric distribution.⁹ That is, for $X = [S \cap C]$,

$$P(X = k) = \frac{\binom{[S]}{k} \binom{[F] - [S]}{[C] - k}}{\binom{[F]}{[C]}}, k = 0, 1, 2, \dots$$

Now let $p = [S]/[F]$ and $q = 1 - p$:

- $E(X) = [C]p = [C]2^{(2^n - s)}/2^{(2^n)} \leq 2^c 2^{(2^n - s)}/2^{(2^n)} = 2^{c-s}$,
- $V(X) = [C]pq([F] - [C])/([F] - 1) \simeq [C]pq \leq 2^{c-s}(1 - 2^{-s})$ for large $[F]$ and $V(X) \simeq 2^{c-s}$ for large S .

Therefore, when $s \geq c$ we expect C and S to share only one function, namely f (which in this case must equal g).

When $s < c$ we would expect f and g to be in the same relatively small subset. This means that there is some chance of g equalling f . However, we cannot explain why, when g is not exactly f , g tends to be closer to f than chance. Apparently the cost neighborhood of a function and the geometric neighborhood of a function are related (i.e. they overlap by more than chance).¹⁰

⁹Mike Breen recognized the relationship between this problem and the hypergeometric distribution.

¹⁰The parity function is an exception to this. That is, for the parity function, the error is worse than chance until there are sufficient samples to extrapolate the function exactly. Also, the Neural Net had trouble with the parity function.

It seems that S is controlled by "geometric" distance while C is controlled by cost differences. What is the relationship between cost and geometric distance and how does this relate to FERD? We first develop an upper bound on cost distance as a function of geometric distance. Second, we draw some conclusions about the "topology" of the space of functions with respect to cost. Finally, we re-interpret some earlier experimental results in a way that applies to this problem.

We can relate the geometric and cost distance between two functions by, in effect, using the computation of one function to compute the other. That is, we compute the more costly function by first computing the less costly function and then computing the differences between the two functions. Define the geometric distance (d_g) between two Boolean functions (f and g) on n variables as:

$$d_g(f, g) = \sum_{x \in X} |f(x) - g(x)|.$$

This is equivalent to the number of points in which f and g differ. Define the cost distance (d_c) between f and g as:

$$d_c(f, g) = |DFC(f) - DFC(g)|,$$

where $DFC(f)$ means the Decomposed Function Cardinality of f .

Theorem 6.9 *If $d_g(f, g) \leq p$ then $d_c(f, g) \leq 4np$.*

Proof:

Assume that $d_g(f, g) \leq p$. Suppose f is the less expensive of the two functions. We can compute g by first computing f (with cost $DFC(f)$), computing the p minority elements where f and g differ (with cost $4(n-1)$ each, or a total of $4p(n-1)$) and then summing f and the p minority elements together (with cost $4(k-1)$, where $k = p+1$ is the number of variables to be summed, for a total cost of $4p$). The DFC of g therefore must not be greater than the cost of the above computation, which is $DFC(f) + 4p(n-1) + 4p = DFC(f) + 4np$. That is, $DFC(g) \leq DFC(f) + 4np$.

□

Theorem 6.10 *If $d_g(f, g) \geq 2^n - p$ then $d_c(f, g) \leq 4np + 2$.*

Proof:

Note that $d_g(f, g) \geq 2^n - p$ implies $d_g(f, \text{not}(g)) \leq p$, since if f and g disagree on all but p points then f and $\text{not}(g)$ agree on all but p points. We can compute $\text{not}(g)$ as above and then complement it with a cost of 2.

□

If neither f nor g have cost ≤ 2 then Theorem 6.10 has the slightly stronger form of Theorem 6.9: if $d_g(f, g) \geq [f] - p$ then $d_c(f, g) \leq 4np$, since the cost of a function with cost more than 2 is the same as its complement.

Theorem 6.9 tells us that functions within a local geometric neighborhood are also within a local cost neighborhood. The converse of Theorem 6.9 is not true. Theorem 6.10 points out a certain kind of symmetry in the cost of functions as you go from one point in the d_g space of functions to its opposite point. The relationship between d_g and d_c is analogous to the temperature distribution on the earth. The temperature everywhere inside this county is within a degree or two of the temperature here. However, when you consider the places that have temperatures about the same as here, you might have to include places as far away as Europe or Asia. We think that the set of low cost functions form many small separated regions within the geometric space of functions. We can think of the learning situation as one where S specifies a fairly large connected geometric region and C specifies a bunch of small separate geometric regions. The geometric regions formed by S are "sub-spaces." For example, if F were 3-D then S might be a plane or a line. The intersection of S and C then might typically be just a sub-space of one of these small pockets of low cost. FERD, in effect, picks one of the lowest cost elements from S intersect C . Looking at the learning curves, we see that as more and more samples are provided the minimum cost function has higher and higher cost. Also, when the cost of the minimum cost function gets up to the cost of f , f is almost always the function FERD selects. It seems the only way there would be more than one function of minimum cost is if the sub-space defined by S runs along a constant cost line (surface or whatever) rather than crossing it. Apparently the odds of that happening are not great.

The expected size of S intersect C is 2^{c-s} . This result depends upon the assumption that S and C are randomly and *independently* selected from F . Let us examine that assumption. Independence requires the distribution of functions with respect to cost within S to be the same as within F as a whole (as in $P(C | S) = P(C)$). The first part of this section showed that within a small d_g neighborhood independence does not hold. However, S is not small and its not really a neighborhood either. We know that all the functions in S are the same on the sample set so their d_g is no more than $[f] - s$. However, in real world learning applications, s is very small compared to $[f]$. That is, S is very large. For any function f_1 in S there exists a function f_2 , not in S , such that $d_g(f_1, f_2) = 1$; that is, they differ only on some point in the sample set. That is, S is not what we would think of as a neighborhood. Therefore, Theorem 6.9 does not preclude independence between C and S for realistic situations.

There is some evidence suggesting that S and C are sufficiently independent. This evidence comes from the minority element experiments (Section 6.2.3). We studied the relationship between the number of minority elements in a function and the function's DFC. It appears that if the number of minority elements is greater than about 20 percent of the size of f then the cost distribution is random (at least the average DFC is about the same as a random distribution). The number of minority elements in a function is the same as its d_g from a constant function. Also, since the DFC of a constant function is zero, the DFC of a function is the same as its d_c from a constant function. Therefore, the DFC versus number-of-minority elements data can be thought of as d_g versus d_c data relative to a constant function. If f is a constant

function and s is less than 60 percent of $[f]$ then S looks pretty much like the set of random functions with greater than 20 percent minority elements. Therefore, if s is less than 60 percent of $[f]$, which it almost always will be, then the cost distribution on S is pretty much random. This means that S and C are reasonably independent under the expected conditions. Intuitively, you would think that you would have to get further from a constant function than any other function before things look random. Therefore, this conclusion that S has a random cost distribution relative to constant functions is even more likely to be true for other functions.

6.6.5 Summary

In summary, FERD is an approach to function extrapolation that chooses the least computationally complex function that is consistent with the samples. In effect, FERD chooses a function from the intersection of the set of low cost functions (C) and the set of functions consistent with the samples (S). Under an assumption of independence of C and S , the expected value of $[C \cap S]$ is 2^{c-s} . We see some evidence that this assumption may not be too bad in practice. Anecdotal comparisons of FERD and Neural Nets suggest that FERD has considerable potential as an extrapolation method. PT 2 will focus on FERD.

6.7 Summary

This chapter reports on the results of looking at the world from a Pattern Theory perspective. We found that randomly generated functions have high DFC and that a very wide range of patterned functions (numeric, symbolic, string, graphic, images and files) have low DFC. There is high correlation between pattern-ness as measured by DFC and as perceived by people. There is also high correlation between DFC and the compression factor for files. These results support the contention that DFC is a measure of the essential pattern-ness of a function. We also found the function decomposition has remarkably general extrapolative properties. This further supports the idea of a fundamental importance of decomposition.

Chapter 7

Conclusions and Recommendations

The theory of computing, as taught in most universities, fails the engineer in several ways. First, it makes much ado over something that is practically trivial, i.e. that there exists *infinite* functions that cannot be computed with *finite* machines (Appendix A). Second, the treatment of complexity results in a statement that is practically not acceptable; i.e. that all finite functions have the same complexity ($O(1)$ time complexity) (Section 4.4). Finally, and most importantly, the theory does *not* support design.

This report introduces a computing paradigm that we hope will support the engineering requirements. Note that most of its elements (the decomposition condition [4], the relationship between combinational and time complexity [64], the theory of program length [12]) are known in the computing theory community, but lack the deserved emphasis. If this report offers anything new to computing theory it is the idea of Decomposed Function Cardinality (DFC) as a general measure of essential computational complexity. DFC correlates very well with the intuitive expectations for a general measure of pattern-ness. We ran the AFD program on a wide variety of patterned functions. Recall that randomly generated functions do not decompose with high probability. Therefore, it is quite remarkable that of the roughly 850 functions that we did not generate randomly, only about 20 did not decompose. We feel that the generality of DFC as a measure of pattern-ness has been well demonstrated.

We can also start to see the potential for practical benefits from Pattern Theory. We have been working with what might be called "toy" problems (i.e. binary functions on no more than 10 or so variables). However, we believe that we should expect to be able to do algorithm design on toy problems before more general problems. Note that this is opposite from the approach taken in most machine learning paradigms. The common view in machine learning is that we solve the easy problems by hand and when we get to problems that we cannot solve by hand such as the character recognition (cannot solve the definition problem) or the traveling salesman problem (cannot solve the realization problem) then we try to get a machine to solve it. We think that we should be able to machine learn (or machine design algorithms for) easy problems

long before we can machine learn the hard ones. Each time the AFD program found that a function had a low DFC (which it did some 830 times), it also found an *algorithm* for that function. Therefore, although we were limited to toy problems, we have demonstrated machine designed algorithms in a very general setting. From a machine learning perspective, FERD performed as well as a Neural Net on functions well suited to Neural Nets and considerably outperformed the Neural Net on other functions. Also, unlike most traditional machine learning paradigms, we can say exactly what property a function must have to be amenable to FERD extrapolation. That property is low computational complexity in the sense of DFC. Recall that DFC is a strikingly general sense of complexity and we begin to appreciate the significance of FERD. Similarly, from a data compression perspective, function decomposition performed comparably to hand-crafted compression algorithms on typical files and considerably outperformed them on atypical files. Therefore, we feel that Pattern Theory represents a solid foundation for an engineering theory of computing.

The results of PT 1 clearly support the need for further exploration and development of Pattern Theory. Although PT 1 dealt with toy problems, we went from a general statement of the real problems to the PT 1 problem in a series of deliberate partitions and simplifying assumptions. Therefore, our recommendations for future developments of Pattern Theory are simply to revisit each decision in defining the PT 1 problem and consider the implications of not making that decision.

Chapter 8

Summary

Algorithm technologies are very important to the Avionics Directorate because they offer a cost-effective means to realize adaptive and fault tolerant avionics capabilities. There is an un-met need for a theory of algorithm design. We only have to consider the power of Estimation Theory or Control Theory to realize the importance of an engineering theory. This report proposes and discusses "Pattern Theory" as a basis for an engineering theory of algorithm design.

Pattern Theory (PT) begins with a very general statement of the algorithm design problem. This general problem covers virtually every other discipline that results in a computational system, including Neural Networks (NN) and model-based vision. Deliberate specializations to this general problem are made to arrive at the problem treated in this report (the PT 1 problem). It is important (and unusual) to state up-front how a specialized approach relates to the general problem.

The problem of finding a pattern in a function is the essence of algorithm design. The key to PT is its measure of pattern-ness: Decomposed Function Cardinality (DFC). The thesis is that low DFC indicates pattern-ness. This measure is uniquely general in reflecting the low complexity that is associated with patterns. The properties of DFC are defined and developed with mathematical rigor. The principal result of PT 1 is a demonstration of the generality with which DFC measures pattern-ness.

The generality of DFC is supported theoretically by 57 proven theorems. DFC has the property that if a function is computationally non-complex relative to time complexity or circuit complexity then it is necessarily non-complex with respect to DFC, while the converse is not necessarily true. The previous statement is not true relative to program length since one can always assign an arbitrarily short program to any function. However, DFC has the property that its average is no more than a single bit greater than the average program length and, when decompositions are encoded as programs, the program length cannot be large without DFC also being large. By rigorously relating DFC to time complexity, program length and circuit complexity it can be assured that the class of patterns defined by DFC includes the classes of patterns that would be defined by these conventional measures. In doing this, new results were also developed in the theory of program length (Appendix A).

PT 1 explored function decomposition algorithms as a means to find DFC pat-

terns. We developed a general test (the Basic Decomposition Condition), based on DFC, for whether or not a function will decompose for a given partition of its variables. This test was then used in algorithms (collectively known as the Ada Function Decomposition (AFD) algorithms) with a variety of heuristics for limiting the depth of the search through iterative decompositions. AFD produces a decomposition (i.e. an algorithm in combinational form) and the DFC of a function. We did not find the hoped for threshold where additional searching had no payoff. However, we did find that the most restrictive search method came very close to performing as well as the least restrictive search method, despite the several orders of magnitude greater run-time of the less restricted search. There was especially little benefit in the larger search when the function being decomposed was highly patterned.

The generality of DFC was also supported experimentally. Over 800 non-randomly generated functions were tested including many *kinds* of functions (numeric, symbolic, string based, graph based, images and files). Roughly 98 percent of the non-randomly generated functions had low DFC (versus less than 1 percent for random functions). The 2 percent that did not decompose were the more complex of the non-randomly generated functions rather than some class of low complexity that AFD could not deal with. It is important to note that when AFD says the DFC is low, which it did some 800 times, it also provides an algorithm. AFD found *the* classical algorithms for a number of functions.

Some applications demonstrate the importance of DFC's generality. The correlation coefficient between DFC and a ranking of patterns by people was 0.8. In a data compression application on typical files, the correlation coefficient between DFC and the compression factor of two commercial data compression programs was about 0.9. However, on an atypical file, AFD had the much better compression factor of 0.04 versus 0.86 and 0.94 for the commercial programs. In a machine learning application, AFD did as well as a NN on problems well suited to NN's. However, on another problem, AFD learned a 128 point function from 30 samples whereas the NN required all 128 points. These applications demonstrate that traditional paradigms look for a particular kind of pattern; when they find that pattern they do well and when they don't they do poorly. PT is unique in that it does not look for a particular kind of pattern, it looks for patterns in general.

PT is a pervasive technology and, at maturity, may revolutionize the approach to many computational problems. Potential areas for early application of PT have been identified, such as algorithm development for Non-Cooperative Target Identification. PT has also shown promise in machine learning and data compression problems. Although PT 1 has laid the foundation and even identified potential early applications, there are many unsolved problems. PT 2 will begin to address the issues that limit the application of decomposition to small problems.

Appendix A

Program Length and the Combinatorial Implications for Computing

A.1 Mathematical Preliminaries

A.1.1 Basic Definitions

A **partial function** $f : X \rightarrow Y$ is a set of ordered pairs from $X \times Y$, where there is at most one $y \in Y$ associated by f with any $x \in X$. That is, $(x, y) \in f$ and $(x, y') \in f$ implies that $y = y'$. A **total function** is a partial function with the additional property that f is defined on all of X . That is, for all $x \in X$, there exists a $y \in Y$ such that $(x, y) \in f$. We will also use the word "mapping" for a function. The set X is called the **domain** of f . The set Y is called the **codomain** of f . The first element of all the pairs in f form a set called the **base** (this is not standard). The base of f is a subset of the domain of f . If f is total then the base of f equals the domain of f . The second element of all the pairs of f form a set called the **range**. The range is a subset of the codomain. The **image** of an element x in the domain of f is the corresponding element $y = f(x)$ in the range of f . When the range of a mapping equals the codomain of the mapping, the mapping is said to be **surjective** or "onto." **Number theoretic functions** are functions of the form $f : N^n \rightarrow N$, where N is the set of natural numbers.

The **cardinality** of a set A , denoted $|A|$, is the number of elements in the set A if A is a finite set. The cardinality of an infinite set is determined by which standard set it corresponds to in a one-to-one manner. The cardinals associated with infinite sets are called **transfinite cardinals**. Since a function f is also a set, $|f|$ is well-defined and indicates the size of a function. Let c indicate the cardinality of the continuum (i.e. the real numbers) and \aleph_0 indicate the cardinality of a countably infinite set. We know that n^{\aleph_0} equals c for any finite n greater than one [20, p.155 Equation 2]. In general, a finite number to the power of a transfinite cardinal is the next larger

cardinal. We have a particular interest in the cardinality of sets involving the Natural numbers (N). All elements of N are finite. However, N is unbounded. That is, there does not exist a $b \in N$ such that $n < b$ for all $n \in N$. Therefore, N has cardinality \aleph_0 and a function with domain N , also has cardinality \aleph_0 . The set of all possible functions with domain N has cardinality c [59, p.12]. The relationship " $<$ " can be interpreted in a natural way on N and on transfinite numbers; i.e. $n < \aleph_0 < c$ for all $n \in N$. That is, the elements of N have their usual order, the transfinite cardinals have their usual order and all transfinite cardinals are greater than all finite cardinals. The relationship " $=$ " also has a natural interpretation, so " \leq " can be used as well.

Let Σ be a set of symbols, not necessarily of finite cardinality. Let Q be N or a subset of N of the form $\{n \in N \mid n \leq T\}$ for some $T \in N$. A string on Σ is a mapping from Q into Σ for some Q . Define Σ^* as the set of all possible strings on Σ . For x , a string in Σ^* , we denote the length of x by $s(x)$. We have not adopted the traditional finite limitation on $[\Sigma]$ and $s(x)$. A language over Σ is a subset of Σ^* .

Let X_1, X_2, \dots, X_n be arbitrary sets. An element of a product of sets (e.g. $X_1 \times X_2 \times \dots \times X_n$) is a vector. A vector is also a string with alphabet $\Sigma = X_1 \cup X_2 \cup \dots \cup X_n$. Therefore, we designate the length of a vector $s(x)$ as its length as a string. For a vector in $X_1 \times X_2 \times \dots \times X_n$, $s(x) = n$.

A.1.2 Combinatorics

Let us review some basic combinatorics. We are especially interested in the cardinality of the set of all vectors of a given length, the set of all strings no longer than a given length, and the set of all functions on a given domain. The following theorem gives the cardinality of a set of vectors.

Theorem A.1 *If V is the set of all vectors of length $s(v) = n$, that is $V = \Sigma^{s(v)}$, then there are $[\Sigma]^{s(v)}$ vectors in V , i.e. $[V] = [\Sigma]^{s(v)}$.*

Proof:

There can be any one of $[\Sigma]$ symbols in the first position. Then for each symbol in the first position, there can be any one of $[\Sigma]$ symbols in the second position. Thus, the number of combinations for a two-dimensional vector is $[\Sigma][\Sigma] = [\Sigma]^2$. Similarly, for each combination of the first two symbols there can be any one of $[\Sigma]$ symbols in the third position. Thus, the number of combinations for a three-dimensional vector is $[\Sigma]^2[\Sigma] = [\Sigma]^3$. This argument can be continued to find that there are $[\Sigma]^{s(v)}$ combinations of symbols in an $s(v)$ -dimensional vector.

□

We now develop the cardinality of a set of strings on an alphabet Σ of $[\Sigma]$ letters.

Theorem A.2 *If L is the set of strings on Σ whose lengths are less than or equal to some threshold $s(l')$ then $[L] = \frac{[\Sigma]^{s(l')+1} - 1}{[\Sigma] - 1}$.*

Proof:

If L is a subset of Σ^* . Let l' be a string in L of the maximum allowed length. There are strings of length $0, 1, 2, \dots, s(l')$. For each string of length i there are $[\Sigma]^i$ different strings (as with vectors). Therefore, there are a total of

$$\sum_{i=0}^{s(l')} [\Sigma]^i$$

strings in L . Thus,

$$\begin{aligned} [L] &= \sum_{i=0}^{s(l')} [\Sigma]^i = \frac{([\Sigma] - 1) \sum_{i=0}^{s(l')} [\Sigma]^i}{[\Sigma] - 1} = \frac{\sum_{i=0}^{s(l')+1} [\Sigma]^i - \sum_{i=0}^{s(l')} [\Sigma]^i}{[\Sigma] - 1} \\ &= \frac{\sum_{i=0}^{s(l')} [\Sigma]^i + [\Sigma]^{s(l')+1} - [\Sigma]^0 + \sum_{i=0}^{s(l')} [\Sigma]^i}{[\Sigma] - 1} = \frac{[\Sigma]^{s(l')+1} - 1}{[\Sigma] - 1}. \end{aligned}$$

□

The expression for the number of vectors is simpler than the expression for the number of strings, yet there is not much difference between the two.

Theorem A.3 For finite $s(l')$ and $s(l') = s(v)$, we have $[V] < [L] < 2[V]$.

Proof:

The first inequality follows from V being a proper subset of L . The second inequality can be developed as follows:

$$[\Sigma]^{s(l')+1} > [\Sigma]^{s(l')+1} - 1 = [\Sigma][\Sigma]^{s(l')} - 1 \geq 2[\Sigma]^{s(l')} - 1,$$

assuming $[\Sigma] \geq 2$. Thus,

$$\begin{aligned} [\Sigma]^{s(l')} + 1 &> 2[\Sigma]^{s(l')} - 1, \\ [\Sigma]^{s(l')} + 1 - 2[\Sigma]^{s(l')} &> -1, \\ 2[\Sigma]^{s(l')} + 1 - 2[\Sigma]^{s(l')} &> [\Sigma]^{s(l')+1} - 1, \\ 2[\Sigma]^{s(l')}([\Sigma] - 1) &> [\Sigma]^{s(l')+1} - 1, \\ 2[\Sigma]^{s(l')} &> ([\Sigma]^{s(l')+1} - 1)/([\Sigma] - 1). \end{aligned}$$

Finally, since

$$[V] = [\Sigma]^{s(v)} = [\Sigma]^{s(l')}$$

and

$$[L] = ([\Sigma]^{s(l')+1} - 1)/([\Sigma] - 1),$$

we have $2[V] > [L]$.

□

Also, for transfinite $s(l')$, we have $[L] = [V]$; i.e. the addition, subtraction or multiplication of a finite number with a transfinite cardinal yields the same transfinite cardinal. Therefore, for finite or transfinite $s(l')$, $[L]$ and $[V]$ are very similar. Let F be the set of all total functions of the form $f : X \rightarrow Y$, where X and Y are arbitrary sets. A set of strings satisfies the prefix condition if no string in the set contains another string from the set as its first letters.

Theorem A.4 *The largest number of different strings of length $\leq l$ and satisfying the prefix condition is the number of vectors of length l .*

Proof:

Any string of length $< l$ eliminates all longer strings with that beginning from the set of strings satisfying the prefix condition. There is always more than one string eliminated by a string of length $< l$. A string of length l only eliminates itself. Therefore, the largest set of strings satisfying the prefix condition is the set of strings of length exactly l .

□

The following theorem gives the cardinality of a function.

Theorem A.5 *If F is the set of all total functions of the form $f : X \rightarrow Y$ then $[F] = [Y]^{[X]}$.*

Proof:

Each function $f \in F$ can be thought of as a vector made up of the sequence of images under f for all $x \in X$. Thus, the cardinality of F is the cardinality of a set of vectors of length $[X]$ over the symbol set Y .

□

We might have assumed that F is the set of all partial (rather than total) functions on a given domain. However, we can model any partial function $g : X \rightarrow Y$ with a total function $f : X \rightarrow (Y \cup \{y'\})$, where y' is some symbol not in Y . That is, let $f(x) = g(x)$ for any x in the base of g and let $f(x) = y'$ for all other x 's. Therefore, the set of all partial functions into Y has a one-to-one correspondence with the set of all total functions into $Y \cup \{y'\}$. The number of partial functions into Y equals the number of total functions into $Y \cup \{y'\}$. The number of total functions into $Y \cup \{y'\}$ is $([Y] + 1)^{[X]}$. The number of partial functions into Y is then $([Y] + 1)^{[X]}$. Therefore, whether we use F as the set of all total functions or as the set of all partial functions, there is little difference in the essential combinatorics.

A.2 Program Length Constraints for Computation

A.2.1 Introduction

In most texts on Computability Theory you will find a statement such as: there exist number theoretic functions (i.e. functions on finite numbers) which cannot be computed with any finite program. The proof of this statement depends only upon some combinatorial arguments. While this is clearly a most impressive consequence of these combinatorial considerations, the consequences are much broader than indicated by that one statement. The objective of this appendix is to recognize and precisely state the general computing limitations implied by combinatorial considerations. As compared with the traditional statement of noncomputability, the following statements are stronger and more general, yet equally easy to derive.

Now it is reasonable to ask whether or not there exists a machine for a given P and F . We know from above that, regardless of the particular association desired between P and F , these simple combinatorial constraints must be satisfied. If they are not satisfied then we say that no machine exists which can be programmed from P to compute any element of F . In particular, if we consider $P = \Sigma_P^*$ then $[P]$ is \aleph_0 . If we also consider F to be the set of all functions on the set of all finite strings then, although $s(x)$ is finite, $[X]$ (or $[f]$) is \aleph_0 and $[F]$ is c . Therefore, $[P]$ is not greater than or equal to $[F]$ and, consequently, there does not exist a machine which can be programmed from P to compute any element of F . This is the traditional statement of noncomputability.

We feel that the generalized developments of this appendix allow a more intuitive introduction to computability as well as the more general implications of program length constraints. That is, first develop the max-min program length constraints. These constraints are intuitive and easy to develop from basic combinatorial considerations. With an understanding of these constraints, we can easily derive the traditional noncomputability result as well as many other more general or stronger results. The existence of noncomputable functions is uncoupled from the traditional sources of confusion such as the relationship between transfinite cardinals and the rich (but irrelevant) structure of Turing machines and number theoretic functions. Therefore, although this alternative introduction to noncomputability is more general and stronger, it can also be more intuitive. We can then develop average-minimum program length constraints by applying Information Theoretic concepts to this paradigm. There are several practically important applications of these program length constraints.

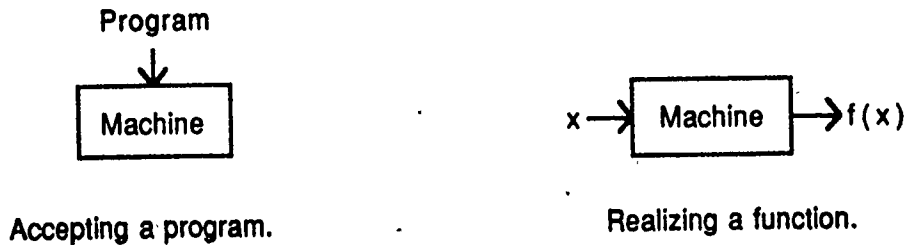


Figure A.1: A Machine's Interfaces

A.2.2 Programmable Machines

Definition of a Programmable Machine

The objective of this appendix is to generalize the development of the combinatorial implications on computing. One area of extended generality is our model of a machine. This section will define our machine model. Many computational systems are well modeled (in terms of the combinatorics of interest) by this machine, including traditional programmable systems, analog computing, logic circuit design and machine learning. Therefore, the theoretical developments with respect to this model have far ranging applications; yet this model is adequate for the many results based on combinatorial considerations of the functions and programs involved. To say that a very general M exists does not tell us much, but, we will be specifying conditions under which M does not exist. Therefore, the generality of the machine model makes the non-existence results stronger.

A **programmable machine** (PM) consists of a programming language P , a set of functions F , and a surjective mapping $M : P \rightarrow F$, i.e. $PM = \{P, F, M\}$. When we say "machine" we mean a closed physical system (i.e. a black box) which may include people. When we say "PM" we mean a model of the physical machine. A **program** is a string from an alphabet Σ_P which satisfies the prefix condition. The prefix condition says that a program p cannot contain any other program as its first i characters, $i = 1, 2, \dots, s(p)$. This condition is necessary and sufficient to make it possible for a machine to be able to decide, after accepting each program character, whether or not a complete program has been accepted. Therefore, P is a subset of Σ_P^* and is formally a language. Of course P could be a traditional programming language (e.g. Fortran); however, it can also be a circuit specification, data samples, etc. The **length of a program** (p) is its length as a string, i.e. $s(p)$.

A machine interfaces with the rest of the world through three avenues (Figure A.1). A machine has the ability to accept a program and realize a function. A machine **realizes** a function if when presented with any element from the function's domain, the machine automatically (i.e. without any help from outside the machine) produces the corresponding element from the function's range. The idea that functions are

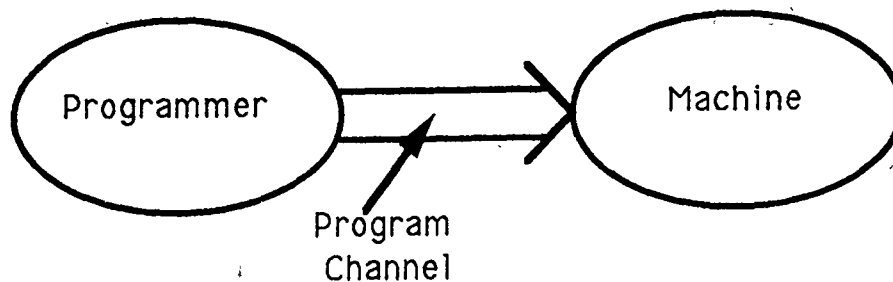


Figure A.2: "Programs" in a Communications Context

an especially important model of computational processes is well developed in the Computing Theory literature (e.g. [50, 59]), so we do not redevelop it here. However, we point out that nearly all computational systems realize a function. A machine has accepted a program if it then realizes the function associated with that program.

We can think of a "program" as a message sent to a machine instructing it to realize a particular function (Figure A.2). The machine is capable of realizing any one of the functions in F when given the appropriate message from P . A program is everything necessary to specify a particular function with respect to a machine capable of realizing a variety of functions. As a "program" crosses this communications channel we have an excellent opportunity to characterize its size. This communications perspective of a program proves very useful in the section on Average-Minimum program length.

Programmable Machines are a very general model of computational systems. A PM is an adequate model of many (perhaps all) computational systems because the semantics of the programming language, which is how computational systems differ, are unimportant with respect to the combinatorics. PM's include traditional programmable systems (as in a general purpose digital computer) and other computational systems that are not obviously "programmable." We demonstrate the idea of a PM with five examples. For each example, we discuss P , F , M , the meaning of "program length" within the context of the example, and how the prefix condition is satisfied.

Examples of Programmable Machines

Random Access Memory (RAM) has a natural PM model. Consider a simplified model of RAM consisting of n address lines, m data lines and a Read/Write line (Figure A.3). A "program" p for the RAM might consist of a sequence of Address-Data combinations with a "Write" value on the Read/Write line, e.g. $p = (d_1, d_2, \dots, d_k)$ where $d_i = \{\text{Write}, \text{Address}_i, \text{Data}_i\}$. P for the RAM might consist of all possible such p 's. A function f realized by a RAM maps Read-Write \times Address into Data, i.e. $\{\text{Read}\} \times \{0, 1\}^n \rightarrow \{0, 1\}^m$, and F might be the set of all possible such functions. M defines f from p , that is, $f(x) = \text{Data}_i$ if there exists a $d \in p$ such that the Address

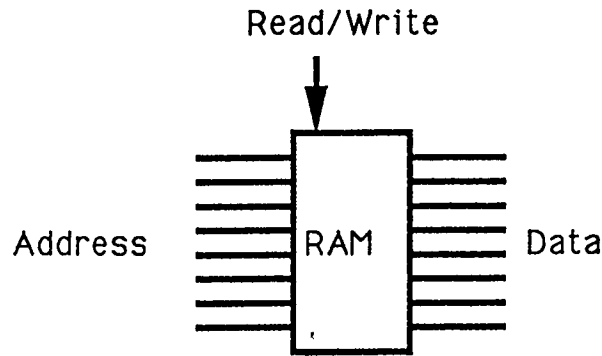


Figure A.3: Simplified RAM Model

of d is x and the *Data* of d is $Data_i$. Program length (in bits) with respect to RAM is the number of bits required to specify a program. For $p = (d_1, d_2, \dots, d_k)$ and $d_i = \{Write, Address_i, Data_i\}$, each d_i consists of 1 bit for Read/Write plus n bits of address plus m bits of data. There are k d_i 's; therefore, there are $k(1 + n + m)$ bits in a program. For total functions, $k = 2^n$, so there are $2^n(1 + n + m)$ bits of program (i.e. $s(p) = 2^n(1 + n + m)$). Therefore, we can define P , F , and M (i.e. PM) such that PM is a model of RAM. In this context the theoretical results which we will develop concerning minimum program length have an interpretation as minimum storage requirements.

For our second example of the generality of the PM model we consider Turing machines. Turing machines are standard models in Computing Theory and are known to be equivalent (in terms of computability) to many other models. Define P to be the set of Turing machines. As pointed out in [59, p.295], a Turing machine is completely defined by its transition function. Therefore, in the context of this example, a program is a transition function. F is the set of computable functions. The exact domain and codomain of $f \in F$ are determined by the input and tape symbol sets. M is the function realized by a person or computer capable of 1) taking a transition function and constructing (or simulating) the Turing machine (i.e. accept the program) and 2) running the Turing machine (i.e. realize the function). "Program length" is the size of the transition function. Traditional computing languages (e.g. Fortran) can similarly be modeled.

As a third example of the PM model consider a "learning by example" machine. We can model a learning machine by defining a program p to be a sequence of examples. P is the set of all example sequences of interest. F is the set of all functions which can be learned by the machine. M models the learning system. M accepts a program by "learning" (e.g. forming rules, forming feature vectors, adjusting weights, etc.). M realizes a function in its post-learning behavior. The "program length" is the size of the example set. In this context, the developments about minimum program

length and "programmability" become minimum example requirements and "learnability." This example could be adapted to other learning modes, where whatever it is that the machine learns from acts as the program.

In this example of a PM we connect "program length" with circuit complexity. Define a program to be the specification of a logic circuit. P is the set of all such specifications. F is the set of all functions of the form $f : \{0,1\}^n \rightarrow \{0,1\}$. M is the function realized by 1) a person or automated manufacturing system which "accepts a program" by constructing a circuit per the specifications and 2) the constructed circuit which realizes a function. There are many ways in which a logic circuit might be specified (e.g. $((w \text{ AND } x) \text{ OR } (\text{NOT } y)) \text{ AND } z$). However, any specification must identify the gates (e.g. AND, OR, NOT) and the interconnection of the gates. Therefore, program length (i.e. length of the circuit specification) is related to the complexity of the circuit (i.e. number of gates and number of interconnections). In this context, the developments concerning minimum program length become limits on minimum circuit complexity.

The discussion in the previous example did not depend on the circuits having discrete values. Therefore we could define a PM for analog computing (electronic, acoustical, optical, etc.). Programs would reflect an identification of the components and their interconnection. F would be the set of functions realizable by some specified analog computer.

Tabular Programs

There is a special form of program, for most machines, with the following characteristics. One, every function in F has a representation of this form and two, the length of the program in this form is the same for every function in F . We call a representation of this form a **tabular** representation. A tabular representation is essentially an exhaustive list or table of the function's values. Revisiting the examples above, every RAM program is tabular. That is, every function has a program and all programs are the same length. With respect to a Turing Machine, a transition function which includes a transition for each ordered pair of f (i.e. x and $f(x)$) is a tabular representation. In this case, as with the RAM, the size of the transition function (program length) corresponds to the size of f (i.e. $s(p) \propto [f]$). A tabular representation with respect to a learning machine simply means that the "example set" is an exhaustive list of the ordered pairs of f . That is, there is an example in the training set for each possible input. Again, the size of the example set (program length) corresponds to the size of f . A tabular representation with respect to logic circuits could be constructed using canonical forms. A tabular representation of a function is a kind of the backstop of all representations.

The Set of Minimum Length Programs

As our final comment on machine models we identify a special subset of the set of all programs. A program specifies which element of F that the machine is to realize.

Each program has exactly one function associated with it. However, each function may have zero, one, or more associated programs. That is, a given program can only realize one function, but, a particular function may have any number of programs. Let P_{min} , a subset of P , consist of those elements of P which do not have a shorter program that realizes the same function. That is,

$$P_{min} = \{p \in P \mid s(p) \leq s(q) \forall q \in P \text{ and } M(p) = M(q)\},$$

where $M(p)$ indicates the function realized by p . A largest element of P_{min} is denoted p' , so that $s(p') \geq s(p) \forall p \in P_{min}$. That is, p' is a largest program of P_{min} required to realize any realizable function of F . If $s(p')$ is greater than some threshold (t) then it immediately follows that there exists a program of length greater than t . That is, p' is a program of length greater than t . $s(p')$ will be called the **max-min program length** of set P with respect to M .

A.2.3 Maximum-Minimum Program Length for Finite and Transfinite Sets

The following theorem relates function set and program set cardinalities.

Theorem A.6 *If (M, P, F) is a Programmable Machine then $[P] \geq [F]$.*

Proof:

By definition of a function, each $p \in P$ can be associated with at most one $f \in F$; or a particular program can only compute one function. Also, since M is onto F , there is a (p, f) pair for every $f \in F$. Therefore, for every $f \in F$ there is at least one unique $p \in P_{min}$ and $P_{min} \subseteq P$.

□

This seems trivial. However, it is the basis for all the combinatorics based non-computability results. Now let p' denote the longest program in P_{min} . Now we relate function set cardinality and program length.

Theorem A.7 *If (M, P, F) is a Programmable Machine and the programs are strings on an alphabet Σ_P of $[\Sigma_P]$ letters then $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1} \geq [F]$.*

Proof:

Theorem A.2 demonstrated that the set of all strings of length less than or equal to $s(p')$ is $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1}$. Therefore, any particular set P , with a longest string p' , cannot have more than $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1}$ elements, i.e. $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1} \geq [P]$. This inequality and that of Theorem A.6 gives us $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1} \geq [P] \geq [F]$.

□

Theorem A.8 *If (M, P, F) is a Programmable Machine and the programs are vectors on an alphabet Σ_P of $[\Sigma_P]$ letters then $[\Sigma_P]^{s(p')} \geq [F]$.*

Proof:

Same as Theorem A.6, except replace the expression for the cardinality of a set of strings with the expression for the cardinality of a set of vectors (Theorem A.1).

□

Theorem A.8 may be interpreted as an approximation of the constraint of Theorem A.7. This approximation is correct to within a factor of 2 by Theorem A.3. The discussion will use this simpler and essentially correct expression. We will call " $[\Sigma_P]^{s(p')} \geq [F]$ " the first max-min program length constraint.

What does the first max-min program length constraint imply about the relationship between function cardinality $[f]$ and program length $s(p')$? In general nothing. That is, the function set F and individual elements of F can have any combination of cardinalities. For example, let the individual functions be of the form $\{(i, j)\}$, with $i, j \in N$. In this example, $[F] = \aleph_0$, while $[f] = 1$. On the other hand, suppose we are only interested in two functions, namely $f(x) = x$ and $g(x) = x + 1$, each on the Reals. In this case, $[F] = 2$, while $[f] = c$. Therefore, in general, the combinatorial constraint implies nothing about the relationship between function cardinality and max-min program length.

Let us assume that the functions of interest are exactly all the total functions on a given domain (X) . This assumption creates a relationship between $[F]$ and $[f]$. This allows us to use the combinatorial constraint to relate $[f]$ and $s(p')$.

Theorem A.9 *If (M, P, F) is a Programmable Machine, the programs are strings on an alphabet Σ_P of $[\Sigma_P]$ letters and F is the set of all total functions of the form $f : X \rightarrow Y$ then*

$$\frac{[\Sigma_P]^{s(p')+1} - 1}{[\Sigma_P] - 1} \geq [Y]^{[X]}.$$

Proof:

By Theorem A.4 $[F] = [Y]^{[X]}$. Theorem A.9 then follows immediately from Theorem A.7.

□

Theorem A.10 *If (M, P, F) is a Programmable Machine, the programs are vectors on an alphabet Σ_P of $[\Sigma_P]$ letters, F is the set of all total functions of the form $f : X \rightarrow Y$ and $[Y] = [\Sigma_P]$ then $s(p') \geq [X]$.*

Proof:

$\frac{[\Sigma_P]^{s(p')+1} - 1}{[\Sigma_P] - 1}$ of Theorem A.9 is replaced by $[\Sigma_P]^{s(p')}$ as in Theorem A.8 to get $[\Sigma_P]^{s(p')} \geq [Y]^{[X]}$. Since $[Y] = [\Sigma_P]$, a log of both sides gives $s(p') \geq [X]$.

□

Note that $[X] = [f]$ for total functions. When we are talking about decision problems (i.e. $[Y] = 2$) and a binary alphabet (i.e. $[\Sigma_P] = 2$), this expression applies. Again, by fixing some particulars and approximating the cardinality of a set of strings we can simplify the expression and better reveal the essential relationships. We call this the second max-min program length constraint.

A comparison between function cardinality and program length seems to be especially natural. The representation of a function as a look-up table is always possible, although infinite functions require infinite tables. We can think of a table with a look-up capability as a "program" and the length of this program is exactly that of the function. This is the basis for many of the developments of Chapter 4.

We can think of the length of an input in two ways. The length of the input as the length of a string might be one measure. For functions with inputs from N , the input length might be the value of the input. However, the set N and the set of finite length strings have the same cardinality, therefore it makes no difference which approach we use. Let us treat input length as the length of the string that expresses the input, and denote it $s(x)$. Now what does the combinatorial constraint imply about the relationship between input length and program length? Again, in general nothing. That is, the function set F and an individual element $f \in F$ can have independent cardinalities (as indicated before) and the cardinality of an input x to f can be independent of $[f]$. For the example with individual functions of the form $\{(i, j)\}$, with $i, j \in N$, $[F] = \aleph_0$ while $s(x) = 1$. On the other hand, for the two functions, $f(x) = x$ and $g(x) = x + 1$, each on the Reals, $[F] = 2$ while $s(x) = \aleph_0$. Therefore, in general, the combinatorial constraint implies nothing about the relationship between input length and max-min program length.

We established in Theorem A.9 that, with an assumption about the form of the function set F , we could relate function cardinality $[f]$ to function set cardinality $[F]$. Now we introduce an additional assumption which allows us to relate input length $s(x)$ and function set cardinality $[F]$. Let us assume, as before, that the functions of interest are exactly all the total functions on a given domain X and codomain Y . In addition let us assume that the domain X is the set of all strings on an alphabet Σ_X of $[\Sigma_X]$ letters. The following theorem relates input length and program length.

Theorem A.11 *If (M, P, F) is a Programmable Machine, the programs are strings on an alphabet Σ_P of $[\Sigma_P]$ letters, F is the set of all total functions of the form $f: X \rightarrow Y$ and X is the set of all strings on Σ_X of length less than or equal to $s(x')$ then*

$$\frac{[\Sigma_P]^{s(p')+1} - 1}{[\Sigma_P] - 1} \geq [Y]^{\frac{[\Sigma_X]^{s(x')+1} - 1}{[\Sigma_X] - 1}}$$

or equivalently

$$\frac{[\Sigma_P]^{s(p')+1} - 1}{[\Sigma_P] - 1} \geq [Y]^{\frac{[\Sigma_X]^{s(x')+1} - 1}{[\Sigma_X] - 1}} \forall x \in X$$

Proof:

From Theorem A.2, $[X] = \frac{[\Sigma_X]^{s(x')+1}-1}{[\Sigma_X]-1}$. Theorem A.11 then follows immediately from Theorem A.9.

□

Theorem A.12 *If (M, P, F) is a Programmable Machine, the programs are strings on an alphabet Σ_P of $[\Sigma_P]$ letters, F is the set of all total functions of the form $f : X \rightarrow Y$ and X is the set of all strings on Σ_X of length less than or equal to $s(x')$ then $s(p') \geq [\Sigma_X]^{s(x')}$.*

Proof:

By Theorem A.1 $[X] = [\Sigma_X]^{s(x')}$. The proof then follows from Theorem A.10.

□

Again, the expression of the constraint can be greatly simplified by assumptions and approximations in non-essential areas. Under these assumptions, if a machine can realize all functions with input less than or equal to $s(x')$ then $s(p')$ is greater than or equal to the power of input length, i.e. $s(p') \geq [\Sigma_X]^{s(x')}$. This might also be expressed $s(p') \geq [\Sigma_X]^{s(x)} \forall x \in X$. This is the third max-min program length constraint.

The remainder of this section is a summary of the developments so far. We discussed the important role of simple combinatorics in deciding the max-min program length. In particular, for there to exist a sufficient number of programs, at least one for each function, the longest program must be at least a certain length. Another way to express this is that there must exist a program of a certain length $s(p')$ or longer. We developed three statements of this combinatorial constraint: 1) $[\Sigma_P]^{s(p')} \geq [F]$, 2) $s(p') \geq [f]$, and 3) $s(p') \geq [\Sigma_X]^{s(x)} \forall x \in X$. We called these the first, second and third max-min program length constraints, respectively. We did not assume finite cardinalities in the development of these constraints; therefore, all the constraints are valid for finite and transfinite cardinals.

The first statement says that there must exist a program whose length is greater than or equal to the number of elements in the set of functions that the machine can be programmed to realize. This is the most general and most explicit of the three statements. It is general in that it uses neither of the assumptions required for statements 2 and 3. In particular, the functions and their domains are not assumed to be of a particular form. The first statement is more explicit in that it includes $[F]$ and $s(p')$, which are the driving parameters of the problem. The other parameters (i.e. $[f]$ or $s(x)$) determine $s(p')$ only indirectly through $[F]$, via the assumptions of the second and third statements.

The second statement says that there must exist a program with length greater than or equal to the number of elements in the functions that the machine can be programmed to realize. This statement requires that we assume a function set of a certain form. Therefore, it is not as general as the first statement. However, this statement has an attractive intuitive interpretation. A function is a set of cardinality

$[f]$, therefore, it is reasonable to expect a representation of the function to be of cardinality $[f]$. Furthermore, any function can be represented with a look-up table (although transfinite functions will require transfinite table sizes) and the size of the table is exactly $[f]$.

The third statement says that there must exist a program with length greater than or equal to the power of the longest input string to the functions that the machine can be programmed to realize. This statement requires two assumptions. One assumption concerns the functions' form relative to the domain. The other assumption concerns the form of the domain itself. Therefore, this third statement is the least general of the three. These assumptions provide the necessary link back to $[F]$; however, $[F]$ is now twice removed in this expression. Therefore this third statement is also the least explicit.

Programmability

We have seen how the combinatorial constraint implies that "programs" must be a certain length if we are to realize a certain variety of functions. However, with the machine model of Section A.2.1 we see that "program" has an unusually general meaning. For example, the combinatorial constraint implies that the specification of a circuit must be so long if the specification format is sufficiently general to allow specifying a certain variety of functions. Similarly, the combinatorial constraint implies that a certain minimum number of examples are required if a learning machine is to be capable of learning a certain variety of functions. The constraint even limits the expressability of a natural language in instructing a person in a task.

We would now like to develop the idea of "programmability." We assume that we have a set of functions (F) which we wish to be able to compute and a set of programs (P) which we are capable of generating. The programmability question then is: "Does there exist a Programmable Machine which will map P into F ?" Another, more traditional, way of viewing this problem is: "Does there exist an $f \in F$ which cannot be programmed from P ?" The previous results provide a sufficient condition for the existence of non-programmable functions.

Theorem A.13 *Given a set of programs P and a set of functions F there exist a non-programmable function if $[P] < [F]$.*

Proof:

There are more f 's in F than there are p 's in P and any given p can realize only one f .

□

We can produce several variants of this result using the conditions of Theorems A.8, A.10, and A.12.

Theorem A.14 *If (M, P, F) is a machine, P is the set of strings on Σ_P of length less than or equal to $s(p')$, F is the set of all total functions of the form $f : X \rightarrow Y$ and*

$$s(p') < [X] \log([Y]) / \log([\Sigma_P])$$

then there exist non-programmable functions. Further, if X is the set of all vectors of length $s(v)$ on Σ_X and

$$s(p') < [\Sigma_X]^{s(v)} \log([Y]) / \log([\Sigma_P])$$

then there exist non-programmable functions.

Proof:

Substitute $[F] = [Y]^{[X]}$ (by Theorem A.4), $[P] = [\Sigma_P]^{s(p')}$ (by Theorem A.1) and $[X] = [\Sigma_X]^{s(v)}$ (by Theorem A.1) into Theorem A.13.

□

With a number of additional specializations to the already specialized result of Theorem A.14, we can arrive at the traditional statement of noncomputability. Computability can be defined by thresholds of allowed resources ($s(p')$) and required performance ($[F]$, $[f]$, or $s(x)$). If these two thresholds are set such that the max-min program length constraint is violated (i.e. $K_p^{s(p')} < [F]$) then there exist noncomputable functions. Otherwise, the max-min program length constraint does not imply noncomputability.

The practice in traditional Computability Theory is to set thresholds which require $s(p')$ and $s(x)$ to be finite but they can be any element of an unbounded set. That is, traditional Computability Theory is concerned with the set of functions with domain N , where N can be thought of as a set of strings. The assumptions used in developing the traditional Computability Theory statement of noncomputability are those satisfied in the third max-min program length constraint. Noncomputability then boils down to the fact that the requirement $s(p') \geq K_X^{s(x)}$ for all finite $s(x)$, implies that $s(p') \geq n$ for all finite n , i.e. that $s(p')$ is not finite. Since we allowed all finite $s(x)$, and required $s(p')$ to be finite, the max-min program length constraint is not satisfied. Therefore, we say there exist "noncomputable" functions. Since $s(x)$ is unbounded and f is defined for all x , we know that $[f]$ is unbounded, in particular, $[f] = \aleph_0$ and $[F] = c$. Therefore, the second max-min program length constraint becomes $s(p') \geq \aleph_0$ and the third statement becomes $K_p^{s(p')} \geq c$. As expected, $s(p')$ is not finite.

Theorem A.15 *If P is the set of strings on Σ_P of length less than or equal to some finite $s(p')$, F is the set of all total functions of the form $f : X \rightarrow Y$, X is the set of all strings of finite length then there does not exist a Programmable Machine (M, P, F) .*

Proof:

From Theorem A.11, $\frac{[\Sigma_P]^{s(p')+1}-1}{[\Sigma_P]-1} \geq [Y]^{\frac{[\Sigma_X]^{s(x)+1}-1}{[\Sigma_X]-1}} \forall x \in X$ if (M, P, F) is a Programmable Machine. However, in this theorem $s(x)$ is unbounded, thus the inequality of Theorem A.11 implies that $s(p')$ must be such that the left side is greater than or equal to the right side for all finite $s(x)$; that is, $s(p')$ is infinite. Therefore, a Programmable Machine is inconsistent with the conditions of this theorem. \square

This last theorem is equivalent to the traditional Computability Theory statement, "There exist number theoretic functions which cannot be computed with a finite program." It is interesting to express this in the context of the first and second constraints on max-min program length. The traditional statement as a comparison of function set cardinality to program length (as in the first constraint) is: "In the infinite set of functions with domain N there exists a function which cannot be computed with a finite program." As a comparison of function cardinality and program length, we might express the traditional Computability Theory statement: "there exist infinite functions that cannot be computed with a finite program." So expressed, noncomputability is much more intuitive. The theorems of this section are a generalization of this familiar Computability result. These statements apply for any function type (even continuous functions), machine type, or programming method. Therefore, regardless of the sophistication of the machine, the efficiency of the program, the degree of structure in the functions to be realized, or the length of the function's input, $s(p')$ is bounded by the above constraints.

Summary

We proposed a generalized development of the combinatorial limitations on computability. A machine is a function $M : P \rightarrow F$ from a set of programs onto a set of functions. By definition of a surjective function, $[P] \leq [F]$. If we assume P is a set of strings on an alphabet of K_P letters with no elements longer than $s(p')$ then essentially $K_P^{s(p')} \leq [P]$. Therefore, $K_P^{s(p')} \leq [F]$. If we also assume that F is the set of all functions on X into Y then $[F] = [Y]^{[X]} = [Y]^{[f]}$. Therefore, $K_P^{s(p')} \leq [Y]^{[f]}$. For a binary programming alphabet and decision problems this becomes $s(p') \leq [f]$. Finally, if we also assume that X is the set of all strings no longer than $s(x')$ then $[X] \leq K_X^{s(x')}$. Therefore, $K_P^{s(p')} \leq [Y]^{K_X^{s(x')}}$. Again, for a binary programming alphabet and decision problems this becomes $s(p') \leq K_X^{s(x')}$ or $s(p') \leq K_X^{s(x)}$ for all $x \in X$. Therefore, in order for a machine $M : P \rightarrow F$ to compute any function in F , there must be a program $p' \in P$ whose length is such that $K_P^{s(p')} \geq [F]$, which under additional assumptions becomes $s(p') \geq [f]$ or $s(p') \geq K_X^{s(x)}$ for all $x \in X$.

A.2.4 Average-Minimum Program Length Bound for Finite Sets

We now develop a lower bound for the average length (i.e. the average over F) of programs in P_{\min} when $[F]$ is finite. First we introduce a few new terms. As before, P is a set of programs satisfying the prefix condition on Σ_P . F is a finite set of functions. P_{\min} is the subset of P containing exactly one shortest program for each f in F . M associates a unique f in F with each p in P . Π is a probability measure on F , i.e. $\Pi : F \rightarrow [0, 1]$ and the sum of $\Pi(f)$ over all $f \in F$ is 1. Using this probability measure, we define the average-minimum program length S_F in the natural way:

$$S_F = \sum_{i=1}^{[F]} \Pi(f_i) s(p_i), \text{ where } p_i = M(f_i), i = 1, 2, \dots, [F], \text{ and } p_i \in P_{\min}.$$

Also based on this probability measure, we can define the entropy H of F with respect to Π :

$$H = \sum_{i=1}^{[F]} \Pi(f_i) \log \frac{1}{\Pi(f_i)}.$$

Entropy has units associated with it which depend on the base of the logarithm. For a base of two the units are called bits.

Notice the exact analogy between the situation defined above and that of Source Encoding in Information Theory [23]. Our set of functions $f_1, f_2, \dots, f_{[F]}$ corresponds to the source alphabet a_1, a_2, \dots, a_K of Information Theory. Our program alphabet corresponds to their code alphabet ($D = [\Sigma_P]$). Our minimum program lengths $s(M(f_1)), s(M(f_2)), \dots, s(M(f_{[F]}))$ correspond to code lengths n_1, n_2, \dots, n_K . Our average-minimum program length S_F corresponds to their average code length (\bar{n}). Our prefix condition for programs is a special case of the Information Theory requirement for unique decodability. The lower bound on average-minimum program length proven below is the analogy of the "Source Encoding Theorem" of Information Theory. This theorem was first proven in [56] as "The Fundamental Theorem for a Noiseless Channel." The proofs given below are from [23].

Two lemmas will be needed for the proof of the average-minimum program length lower bound.

Lemma A.1 $\ln(z) \leq z - 1$ for all $z > 0$ and equal if and only if $z = 1$.

Proof:

Consider the first and second derivatives of $\ln(z) - z + 1$.

□

The following lemma is known as the Kraft inequality.

Lemma A.2 If P is a set of programs for F then

$$\sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} \leq 1.$$

Proof:

Define $s' = \max(s(p_1), s(p_2), \dots, s(p_F))$ i.e. s' is the largest program in P_{\min} . Let A be the set of all possible programs of length s' . $[A] = [\Sigma_P]^{s'}$. Each program p_i in P_{\min} corresponds to a subset of A (i.e. A_i) as follows: $A_i = \{a \in A \mid \text{the string made up of the first } s(p_i) \text{ elements of } a \text{ is the same as } p_i\}$. That is, A_i is the set of all strings of length s' which begin with the substring p_i . All of the A_i 's are disjoint because of the prefix condition. That is, if a is in both A_i and A_j then p_i or p_j is a prefix of the other, which violates the prefix condition. Since the A_i 's are disjoint subsets of A ,

$$\sum_{i=1}^{[F]} [A_i] \leq [A].$$

Note that $[A] = [\Sigma_P]^{s'}$ and $[A_i] = [\Sigma_P]^{s'-s(p_i)}$. Therefore,

$$\sum_{i=1}^{[F]} [\Sigma_P]^{s'-s(p_i)} = [\Sigma_P]^{s'} \sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} \leq [\Sigma_P]^{s'}$$

Or,

$$\sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} \leq 1$$

□

The principle result of this section, the average-minimum program length lower bound, can now be stated and proven.

Theorem A.16 *If (M, P, F) is a machine with probability measure Π on F then $S_F \geq H/\log[\Sigma_P]$. That is, the average-minimum program length is greater than or equal to the entropy of F with respect to Π divided by the log of the program alphabet size. The base of the logarithm in the denominator of the right hand side of the inequality is determined by the units of H (i.e. the log used in computing H and that used in the inequality must have the same base).*

Proof:

We show that $H - S_F \log([\Sigma_P]) \leq 0$.

$$\begin{aligned} H - S_F \log([\Sigma_P]) &= \sum_{i=1}^{[F]} \Pi(f_i) \log \frac{1}{\Pi(f_i)} - \sum_{i=1}^{[F]} \Pi(f_i) s(p_i) \log[\Sigma_P] \\ &= \sum_{i=1}^{[F]} \Pi(f_i) \log \frac{1}{\Pi(f_i)} + \sum_{i=1}^{[F]} \Pi(f_i) \log [\Sigma_P]^{-s(p_i)} \\ &= \sum_{i=1}^{[F]} \Pi(f_i) \left\{ \log \frac{1}{\Pi(f_i)} + \log [\Sigma_P]^{-s(p_i)} \right\} \end{aligned}$$

$$= \sum_{i=1}^{[F]} \Pi(f_i) \log \left(\frac{[\Sigma_P]^{-s(p_i)}}{\Pi(f_i)} \right)$$

Let $z = \frac{[\Sigma_P]^{-s(p_i)}}{\Pi(f_i)}$, then

$$H - S_F \log[\Sigma_P] = \sum_{i=1}^{[F]} \Pi(f_i) \log(z) = \log e \sum_{i=1}^{[F]} \Pi(f_i) \ln z$$

From Lemma A.1, $\ln z \leq z - 1$, thus

$$\begin{aligned} H - S_F \log[\Sigma_P] &\leq \log e \sum_{i=1}^{[F]} \Pi(f_i) (z - 1) = \log e \sum_{i=1}^{[F]} \Pi(f_i) \left(\frac{[\Sigma_P]^{-s(p_i)}}{\Pi(f_i)} - 1 \right) \\ &= \log e \sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} - \Pi(f_i) = \log e \left(\sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} - \sum_{i=1}^{[F]} \Pi(f_i) \right) \\ &= \log e \left(\sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} - 1 \right) \end{aligned}$$

From Lemma A.2,

$$\sum_{i=1}^{[F]} [\Sigma_P]^{-s(p_i)} \leq 1 \text{ therefore, } H - S_F \log[\Sigma_P] \leq 0.$$

□

For finite $[F]$, we can quickly establish a generalized form (with respect to Π) of the Max-Min Program Length Lower Bound.

Corollary A.1 *If (M, P, F) is a machine with probability measure Π on F then $s(p') \geq H / \log[\Sigma_P]$.*

Proof:

$$S_F = \sum_{i=1}^{[F]} \Pi(f_i) s(p_i) \leq \sum_{i=1}^{[F]} \Pi(f_i) s(p') = s(p') \sum_{i=1}^{[F]} \Pi(f_i) = s(p')$$

and $S_F \geq H / \log[\Sigma_P]$ by Theorem A.16. Therefore, $s(p') \geq S_F \geq H / \log[\Sigma_P]$.

□

Suppose that all of the functions in F are equally probable; i.e. $\Pi(f) = 1/[F]$ for all $f \in F$.

Corollary A.2 *If (M, P, F) is a machine with a probability measure Π on F such that $\Pi(f_i) = \Pi(f_j)$ for all i, j then $S_F \geq \log[F] / \log[\Sigma_P]$.*

Proof:

The entropy (H) of F is :

$$\begin{aligned} H &= - \sum_{f \in F} \Pi(f) \log \Pi(f) = - \sum_{f \in F} \frac{1}{[F]} \log \frac{1}{[F]} = \sum_{f \in F} \frac{\log[F]}{[F]} \\ &= \frac{\log[F]}{[F]} \sum_{f \in F} 1 = \frac{\log[F]}{[F]} [F] = \log[F] \end{aligned}$$

By Theorem A.16, we have that $S_F \geq H / \log[\Sigma_P] = \log[F] / \log[\Sigma_P]$.

□

Therefore, the average minimum program length is greater than or equal to $\log[F]$. Not only does there exist a program of length $\log[F]$ or longer (Theorem A.7), the average length is $\log[F]$ or longer. The following corollary gives the lower bound with respect to f and $s(x)$.

Corollary A.3 *If (M, P, F) is a machine with equally likely probability measure Π on F and F is the set of all total functions of the form $f : X \rightarrow Y$ then*

$$S_F \geq [X] \frac{\log[Y]}{\log[\Sigma_P]}$$

Further, if X is the set of all vectors of length $s(v)$ on Σ_X then

$$S_F \geq \frac{[\Sigma_X]^{s(v)} \log[Y]}{\log[\Sigma_P]}.$$

Proof:

Substitute $[F] = [Y]^{[X]}$ (by Theorem A.4) into Corollary A.2 to get the first result. Substitute $[X] = [\Sigma_X]^{s(v)}$ (by Theorem A.1) into the first part to get the second part.

□

The following corollary demonstrates that if there is a short program then there must also be a long program.

Corollary A.4 *If (M, P, F) is a Programmable Machine where there exists an $f \in F$ such that $f = M(p_1)$, $p_1 \in P_{\min}$, and $s(p_1) < S_F$ then there exists a g in F such that $g = M(p_2)$, $p_2 \in P_{\min}$, and $s(p_2) > S_F$.*

Proof:

Suppose to the contrary, that is, $s(p_1) < S_F$ but there does not exist g such that $g = M(p_2)$, $p_2 \in P_{\min}$, and $s(p_2) > S_F$. In other words, for all $p \in P_{\min}$, $s(p) \leq S_F$. However

$$S_F = \sum_{i=1}^{[F]} \Pi(f_i) s(p_i) = \Pi(f_1) s(p_1) + \sum_{i=1}^{[F]} \Pi(f_i) s(p_i)$$

```

10 REM A BASICA Program to Demonstrate a Tabular Data Structure
20 DIM F(9)
30 DATA/3,2,7,5,4,7,8,3,6,4/
40 FOR I=0 TO 9
50     READ F(I)
60 NEXT I
70 INPUT X
80 PRINT F(X)
90 GOTO 70
100 END

```

Figure A.4: BASIC Allows for Tabular Data Structures

$$< \Pi(f_1)S_F + \sum_{i=1}^{[F]} \Pi(f_i)S_F$$

since $s(p_1) < S_F$ and, for all $p \in P_{\min}$, $s(p) \leq S_F$. However,

$$\Pi(f_1)S_F + \sum_{i=1}^{[F]} \Pi(f_i)S_F = S_F \sum_{i=1}^{[F]} \Pi(f_i) = S_F.$$

Therefore, $S_F < S_F$, which is a contradiction.

□

The average-minimum program length lower bound of Section A.2.2 combined with the exponential relationship between input size and function size will be used to show that realistically programmable functions are an extremely small fraction of possible functions. At any given time there is some upper bound (B_P) on the length of realistic program lengths. For example, limited by today's technology, there are no programs of lengths greater than say 10^{10} bits. Let B_X be an upper limit on allowed function input sizes, allowing larger inputs would only make the fraction of realistically programmable functions even smaller. Most computer languages allow for the representation of a function in a basically tabular structure (e.g. Figure A.4).

We limit our discussion in this section to machines which allow tabular programs. Any traditional computer language is included. Some machine learning is included, where a "tabular program" is an exhaustive sample set. Some circuit design situations could be included here also. Therefore, while the set of machines which allow tabular programs does not include all machines, it is still a very general computer model.

We want to characterize the size of tabular programs in terms of a "table" and some "overhead." In Figure A.4 the numbers in the DATA statement are the "table" and the rest of the program is the "overhead." We can design a program whose overhead is essentially constant with respect to increasing function size. The example

in Figure A.4 requires changing the DIM and FOR statements (these grow as the log of function size); however, it is possible to design around such statements. Define M' to be the set of machines which allow programs with lengths $\leq [f] + K$, for any f and some constant K . Let F be a set of functions on inputs of length B_X . Let G (the "good" functions) be the subset of F such that

$$G = \{f \in F \mid \exists p \ni f = M'(p) \text{ implies that } s(p) \leq B_P\}.$$

Define $R = [G]/[F]$, the fraction of F with programs of length B_P or less. This fraction of realistically computable functions is very small.

Theorem A.17 For any machine in M' , $R \leq \frac{K}{[\Sigma_X]^{B_X} - B_P + K}$

Proof:

Let S_Φ be the average-minimum program length of the set of functions Φ . We are especially concerned with S_F, S_G and S_{F-G} .

$$\begin{aligned} S_F &= \frac{[G]S_G + [F - G]S_{F-G}}{[F]} \\ &= RS_G + (1 - R)S_{F-G} \end{aligned}$$

Since every function in f has a tabular program of cost $[f] + K$, the average-minimum cost over F (or any subset of F) cannot exceed this. Therefore,

$$S_{F-G} \leq [f] + K = [\Sigma_X]^{B_X} + K$$

and

$$S_G \leq B_P$$

from the definition of B_P . Substituting these relations into the expression for S_F :

$$S_F = RS_G + (1 - R)S_{F-G} \leq R(B_P) + (1 - R)([\Sigma_X]^{B_X} + K)$$

From Theorem A.16, $S_F \geq [f] = [\Sigma_X]^{B_X}$. Therefore,

$$\begin{aligned} [\Sigma_X]^{B_X} &\leq S_F \leq R(B_P) + (1 - R)([\Sigma_X]^{B_X} + K) \\ [\Sigma_X]^{B_X} &\leq R(B_P) + [\Sigma_X]^{B_X} + K - R([\Sigma_X]^{B_X} + K), \\ K &\geq R(B_P - [\Sigma_X]^{B_X} + K). \end{aligned}$$

Therefore,

$$R \leq \frac{K}{[\Sigma_X]^{B_X} - B_P + K}.$$

□

We see that the fraction of 'good' (i.e. realistically programmable) functions goes as the inverse of the difference between the bound in program size (B_P) and the bound

n	$\log R_1$	$\log R_2$
12	-256	> 0
25	-3.3546×10^7	-8.554×10^6
50	-1.1259×10^{15}	-1.1259×10^{15}
100	-1.2676×10^{30}	-1.2676×10^{30}

Table A.1: Fraction of Functions Computable by NN

in function size ($[\Sigma_X]^{Bx}$). However, the bound in function size goes up exponentially with the bound in input size. Therefore, the fraction of 'good' functions is extremely small for any reasonable combination of bounds. For example, even if we say that programs up to 10^{10} bits are realistic, then less than 10^{-12} percent of the functions on 64 bits (e.g. two 32 bit integers) have realistic programs.

There are many questionable particulars in the assumptions of Theorem A.17. For example, the 'overhead' should perhaps be log or linear rather than constant. However, the point remains valid. That is, the fraction of functions which we can reasonably expect to program is extremely small. Functions with reasonable programs are very special. This result suggests that we should not discount representation methods because they only have efficient representations on a small class of functions. For example, one reason often given in Switching Theory for not pursuing Function Decomposition as a design method is that only a small fraction of functions decompose. However, it was just demonstrated that any representation method will only efficiently represent a small fraction of functions.

The limitations of Neural Nets (NN) are quite striking when we interpret the learned parameters of the net as its program.¹ A fully interconnected NN of depth d and n input bits has approximately $d \times n$ parameters which can be adjusted as the net learns. Each parameter is some real variable of say r bits. Therefore, the program length is $r \times d \times n$. The number of programs possible on such a NN is $2^{r \times d \times n}$. The number of functions on n variables is 2^{2^n} . The fraction of computable functions then is $R = \frac{2^{r \times d \times n}}{2^{2^n}}$; which gets very small very fast. R_1 in Table A.1 is for $r = 64$ and $d = 5$, which is a reasonable NN. R_2 is for $r = 1000$ and $d = 1000$, which is much larger than most people are considering for NN's. Note that the table contains the *logarithm* of R . Therefore, only an infinitesimal fraction of functions are computable with any foreseeable NN.

We now demonstrate a machine which is optimal in terms of the average required program length. Assume that the finite set X is ordered. Define a Table Machine as a machine whose program p_i simply lists, in order of X , the images of $f_i : X \rightarrow Y$. Assume that F is the set of all total functions from X into Y , where Y is also a finite set. Assume that the elements of X , when considered as strings, satisfy the prefix condition. Finally assume that all elements of F are equally probable.

¹See [2] for a more sophisticated treatment of this idea.

```

10 REM A Basica Program for a Z-248 as a demonstration of a Table Machine.
20 DIM(9)
30 REM Accept the Program
40 FOR I=0 TO 9
50     PRINT "F(";I;")=";
60     A$=INKEY$: IF A$=' ' THEN 60
70     F(I)=VAL(A$)
80     PRINT F(I)
90 NEXT I
100 PRINT "Press S to stop"
110 REM Realize the Function
120 PRINT "X=";
130 A$=INKEY$: IF A$="" THEN 130
140 IF A$="S" THEN 190
150 X=VAL(A$)
160 PRINT X
170 PRINT "F(X)=";F(X)
180 GOTO 120
190 END

```

Figure A.5: An Example Table Machine

An example of a Table Machine for $X = Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is given in Figure A.5 (actually the Table Machine consists of a PC running this program). This example demonstrates that Table Machines exist in the “real world.”

The “program” for this example is the first 10 digits that one enters in response to the $F(I)=?$ prompts. We now prove that a Table Machine is optimal in terms of average required program length.

Theorem A.18 *If X is an ordered finite set whose elements satisfy the prefix condition, Y is a finite set, F is the set of all total functions of the form $f : X \rightarrow Y$, the elements of F are equally probable, P_T is the set of programs associated with a Table Machine, and P_M is the set of programs associated with any other machine, then $S_{F-T} \leq S_{F-M}$. That is, the average-minimum program length for a Table Machine is optimal.*

Proof:

$s(p_T) = [X]$ for $p_T \in P_T$ from the definition of a Table Machine. $S_{F-T} = s(p_T)$ since all elements of F are equally probable. Therefore, $S_{F-T} = [X]$. For an arbitrary machine M , $S_{F-M} \geq \frac{\log[F]}{\log[\Sigma_P]}$ from Corollary A.3. However, $[F] = [Y]^{[X]}$ since F is the set of all total functions on X into Y and $[Y] = [\Sigma_P]$ from the definition of a Table

Machine. Then,

$$\log[F] = [X] \log[\Sigma_P] \text{ and } S_{F-M} \geq \frac{[X] \log[\Sigma_P]}{\log[\Sigma_P]}.$$

Therefore, $S_{F-M} \geq [X] = S_{F-T}$.

□

One consequence of this theorem is that "powerful" languages (e.g. Fortran is more "powerful" than a machine language) do not lessen the average-minimum program length. The intuition that more powerful languages should allow a simpler expression of a relationship is shown to depend a non-equally probable distribution of the function set. This suggests a connection between language design and some assumed (at least implicitly) probability distribution on F .

Theorem A.19 establishes that the bounds of Theorem A.16, Corollary A.2 and Corollary A.3 are the *greatest* lower bounds when the assumptions of Theorem A.18 are met.

Theorem A.19 *If the conditions of Theorem A.18 are met, then the lower bounds of Theorem A.16, Corollary A.2 and Corollary A.3 are greatest lower bounds of average-minimum program length.*

Proof:

If there were a greater lower bound then it would be violated by a Table Machine.

□

A.3 Summary

This appendix develops a theory of computational complexity based on program length. We formally defined our concept of programmable machine and proved numerous properties of program length. A programmable machine is sufficiently abstract to include many different kinds of problems. Under various interpretations the program length results become memory requirements, learnability, or circuit complexity results. In effect, we have brought into an engineering setting some of the developments of Shannon, Turing, Chaitin and others. This common setting then has extended applications, such as learnability, circuit complexity and especially Pattern Theory.

Appendix B

Function Decomposition Program User's Guide

This software is located in `ficsim::user2:[vogt]`¹. There are 10 (that's right 10) different versions characterized as follows:

- V.1. (8): Non-shared, exhaustive
- V.2. (2): Non-shared, negative decompositions
- V.2A. (0): Non-shared, negative decompositions,
greedy search
- V.2B. (3): Non-shared, negative decompositions,
number of cares cost
- V.2AB. (1): Non-shared, negative decompositions,
number of cares cost, greedy
- V.3. (9): Shared, exhaustive
- V.4. (6): Shared, negative decompositions
- V.4A. (4): Shared, negative decompositions,
greedy search
- V.4B. (7): Shared, negative decompositions,
number of cares cost
- V.4AB. (5): Shared, negative decompositions,
number of cares cost, greedy

The numbers in parentheses indicate relative speed, with V.2A being the fastest and V.3 the slowest. These are only rough estimates, and may even be slightly wrong.

To run the program, get into the directory appropriate to the version you want to run, e.g.

```
$ set def [vogt.pbml.v4ab]
```

and then type

¹This User's Guide was written by Chris Vogt.

run pbml_driver

It's that simple. The program will then come up with the following prompt:

SPECIFY FUNCTION TABLE VALUES:

Enter 0 to input from a file, 1 from terminal,
or 2 to QUIT program:

Interactive Interface

To enter the function in question interactively, enter a 1. The program will then prompt you for the information needed. The prompts are relatively straightforward. A sample session to input a "checkerboard" on 3 variables (i.e. 01011010) is shown below:

Name of function: checkerboard3
How many input variables does the function have? 3
Enter 0 to input falses, 1 to enter trues: 1
How many function values will you be entering? 4
Enter negative values below for Don't Cares
Enter decimal equivalent of binary input that has a true value: 1
Enter decimal equivalent of binary input that has a true value: 3
Enter decimal equivalent of binary input that has a true value: 4
Enter decimal equivalent of binary input that has a true value: 6

To enter "don't care" values, use a negative number. For example, if we wanted to enter the function 010X1010, we would have typed -3 instead of 3.

Input From a File

If you are going to be using a function often, it may be handy to store it in a file. To input a function from a file, type 0 to the first prompt. The program will then prompt you for the file name. Make sure you give the full file specification needed, or the program will crash. The function is stored in the file with the following format: The first $n + 1$ digits indicate the number of input variables in unary. The next 2^n digits are the function itself. Thus, the file for the checkerboard entered above would be:

1
1
1
0
0
1
0
1
1
1

0
1
0

Note that the first 4 digits (1110) tell us that $n = 3$, and the next 8 are the actual function. To indicate a "don't care" in the function, use a 2 instead of a 0 or 1.

Non-interactive Runs/Batch Jobs

Sometimes you may want to make runs of several functions and save the output. This can be done simply using a command file. A sample command file, BATCH-DRIVER.COM, is found in each version's [.TEST] subdirectory. It consists of two lines:

```
$ set def [vogt.pbml.v1.test]  
$ run/detached/input=input.dat/output=output.dat [-]pbml_driver
```

To utilize this file, you must first create a file *input.dat* to be used as input to the program. It should contain exactly what you would type if you ran the program. A sample *input.dat* is shown here:

```
0  
[vogt.pbml.data]CHECK6.DAT  
0  
[vogt.pbml.data]LETTERA.DAT  
0  
[vogt.pbml.data]MIKES_EXAM.DAT  
1  
checkerboard3  
3  
1  
4  
1  
3  
4  
6  
2
```

Note, this input file will first decompose three functions which are specified in files (check6.dat, lettera.dat, mikes-exam.dat) and then will decompose the checkerboard on 3 variables as in the example shown previously. The last 2 in the file is the command to quit the program.

After having created *input.dat*, one runs the program by issuing the command:

```
$ submit/nolog/noprint/notify batch_driver.com
```

The output from the program will be saved in the file *output.dat*, which will not be accessible until the program is done running.

Recompilation

If changes are made to the source code, recompilation is simple. In each version's directory, there is a com file called *comp*.com* which does the job. The * is replaced with the version number. Thus, in [vogt.pbml.v2ab] there is a file called *compv2ab.com*. This file can be executed interactively (with the @ sign), or can be submitted as a batch job.

Two WARNINGS about recompiling:

- At present there is just barely enough disk quota to handle all of the files created during compilation. You need about 1300 blocks free, so check the quota beforehand.
- After compiling, to get rid of all of the "unnecessary" files created, execute the command file *killada.com* found in [vogt.com].

Bibliography

- [1] Thomas Abraham. *Pattern Recognition: Machine vs. Man*. Final Report, USAF-UES Summer Faculty Research Program, July 1990.
- [2] Ysar S. Abu-Mostafa. Information theory, complexity and neural networks. *IEEE Communications Magazine*, 27(11):25-28, 1989.
- [3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.
- [4] Robert L. Ashenurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, April 1957. Also in *The Annals of the Harvard Computational Laboratory XXXIX*, Harvard University Press, Cambridge, Massachusetts, 1959, pages 74-116, and in Curtis'62 pages 571-602.
- [5] Mark Boeke. *Pattern Based Machine Learning*. Final Report, AFOSR High School Apprenticeship Program, August 1990.
- [6] John D. Bransford and Barry S. Stein. *The IDEAL Problem Solver*. W. H. Freeman and Company, New York, 1984.
- [7] Gilles Brassard and Paul Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [8] Mike Breen. *Some Results in Pattern-Based Machine Learning*. Final Report, USAF Summer Faculty Research Program, 1990.
- [9] Richard Burden and J. Faires. *Numerical Analysis*. Prindle, Weber and Schmidt, Boston, third edition, 1985.
- [10] Maureen Caudill. Neural networks primer. *AI Expert*, 3(6):53-59, June 1988.
- [11] Michael Chabinyc. *Pattern Based Machine Learning: A Comparison of Ada Function Decomposer Versions*. Final Report, AFOSR High School Apprenticeship Program, August 1990.
- [12] Gregory J. Chaitin. *Algorithmic Information Theory*. Cambridge University Press, New York, 1987.

- [13] Paul R. Cohen and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 1982.
- [14] H. Allen Curtis. *A New Approach to The Design of Switching Circuits*. D. Van Nostrand Company, Princeton, New Jersey, 1962.
- [15] Lawrence Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann, Palo Alto, California, 1987.
- [16] Pierre A. Devijver and Josef Kittler. *Pattern Recognition: A Statistical Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1982.
- [17] D. Henry Edel. *Introduction to Creative Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
- [18] Michael J. Findler. *Neural Networks and Machine Learning*. Graduate Student Research Program Final Report, Air Force Office of Scientific Research, August 1989.
- [19] M. J. Fischer and N. Pippenger. *M. J. Fischer Lecture Notes on Network Complexity*. Universitat Frankfurt, Frankfurt, 1974.
- [20] A. A. Fraenkel. *Abstract Set Theory*. North-Holland Publishing Company, Amsterdam, 1953.
- [21] Arthur D. Friedman. *Fundamentals of Logic Design and Switching Theory*. Computer Science Press, Rockville, Maryland, 1986.
- [22] King Sun Fu. *Syntactic Pattern Recognition and Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1982.
- [23] R. Gallager. *Information Theory and Reliable Communications*. Wiley, New York, 1968.
- [24] Wendell Garner. Good patterns have few alternatives. In I. Janis, editor, *Current Trends in Psychology*, pages 185-192, William Kaufmann Inc., Los Altos, California, 1977.
- [25] Thomas K. Gearhart. *Investigations of a Lower Bound on the Error in Learned Functions*. Final Report, USAF-UES Summer Faculty Research Program, July 1990.
- [26] Donald D. Givone. *Introduction to Switching Circuit Theory*. McGraw-Hill, New York, 1970.
- [27] Gilbert Held and Thomas Marshall. *Data Compression*. Wiley, New York, second edition, 1987.
- [28] Vladimir Hubka. *Theory of Technical Systems*. Springer-Verlag, New York, 1988.

- [29] Laveen N. Kanal. Patterns in pattern recognition: 1968-1974. *IEEE Transactions on Information Theory*, 20:697-722, November 1974.
- [30] John Langenderfer. *A Study of the Computational Complexities of Functions and their Inverses*. Memorandum, Wright Research and Development Center, WL/AART Wright Patterson AFB, OH, September 1989.
- [31] Pat Langley, editor. *Proceedings of the Fourth International Workshop on Machine Learning*, Morgan Kaufmann, Palo Alto, California, 1987.
- [32] Pat Langley, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Process*. The MIT Press, Cambridge, Massachusetts, 1987.
- [33] Ming Li and Paul M. B. Vitányi. Kolmogorov complexity and its applications. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 189-254, The MIT Press, Cambridge, Massachusetts, 1990.
- [34] O. B. Lupanov. A method of circuit synthesis. *Izv. V.U.Z. Radiofiz.*, 1(1):120-140, 1958.
- [35] M. Machtey and P. R. Young. *Theory of Algorithms*. Elsevier North-Holland, New York, 1978.
- [36] Peter S. Maybeck. *Stochastic Models, Estimation and Control*. Volume 1, Academic Press, New York, 1979.
- [37] Peter S. Maybeck. *Stochastic Models, Estimation and Control*. Volume 2, Academic Press, New York, 1982.
- [38] Loren P. Meissner. *The Science of Computing*. Wadsworth Publishing, Belmont, California, 1974.
- [39] James L. Melsa and David L. Cohn. *Decision and Estimation Theory*. McGraw-Hill, New York, 1978.
- [40] J. L. Meriam. *Statics*. Wiley, New York, second edition, 1971.
- [41] Paul L. Meyer. *Introductory Probability and Statistical Applications*. Addison-Wesley, Reading, Massachusetts, second edition, 1970.
- [42] Ryszard Michalski, Jaime Carbonell, and Tom M. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Volume 1, Morgan Kaufmann, Palo Alto, California, 1983.
- [43] G. A. Miller. The magic number seven, plus or minus two: some limits on our capacity for processing information. *The Psychological Review*, 63:81-97, March 1956.

- [44] Gerald J. Montgomery and Keith C. Drake. Abductive networks. In *SPIE Applications of Neural Networks Conference*, April 1990.
- [45] Saburo Muroga. *Logic Design and Switching Theory*. Wiley, New York, 1979.
- [46] A. Oppenheim and R. Schaffer. *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1975.
- [47] Engineering Concepts Curriculum Project. *Man and His Technology*. McGraw-Hill, New York, 1973.
- [48] Timothy D. Ross. *Elementary Theorems in Pattern Theory*. PhD thesis, Air Force Institute of Technology, 1988.
- [49] Timothy D. Ross. *Pattern Representation and Recognition*. Research Prospectus, Air Force Institute of Technology, 1986.
- [50] Timothy D. Ross and Alan V. Lair. Definition and realization in pattern recognition system design. In *Proceedings of the 1987 IEEE Int. Conf. on Systems, Man and Cybernetics*, pages 744-748, 1987.
- [51] Timothy D. Ross and Alan V. Lair. On the role of patterns in recognizer design. In Josef Kittler, editor, *Pattern Recognition*, pages 193-202, Springer-Verlag, New York, 1988.
- [52] J. Paul Roth. Minimization over Boolean trees. *IBM Journal*, 543-558, November 1960.
- [53] David E. Rumelhart, James L. McClelland, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Volume 1, The MIT Press, Cambridge, Massachusetts, 1986.
- [54] John E. Savage. *The Complexity of Computing*. Wiley, New York, 1976.
- [55] C. P. Schnorr. The network complexity and the Turing machine complexity of finite functions. *Acta Informat.*, 7:95-107, 1976.
- [56] Claude E. Shannon. A mathematical theory of communication. *Bell Systems Tech. Journal*, 27:379-423 (Part I), 623-656 (Part II), 1948. Reprinted in book form with postscript by W. Weaver, Univ. of Illinois Press, Urbana, 1949.
- [57] Harold A. Simon. *A Student's Introduction to Engineering Design*. Pergamon Press, New York, 1975.
- [58] Jean-Claude Simon. *Patterns and Operators: The Foundations of Data Representation*. McGraw-Hill, New York, 1986. Translated by J. Howlett.
- [59] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computation*. Addison-Wesley, Reading, Massachusetts, 1988.

- [60] H. C. Torng. *Switching Circuits: Theory and Logic Design*. Addison-Wesley, Reading, Massachusetts, 1972.
- [61] J. F. Traub, G. W. Wasilkowski, and H. Wozniakowski. *Information-Based Complexity*. Academic Press, New York, 1988.
- [62] Leonard Uhr, editor. *Pattern Recognition: Theory, Experiment, Computer Simulation, and Dynamic Models of Form Perception and Discovery*. Wiley, New York, 1966.
- [63] Satoshi Watanabe. *Pattern Recognition: Human and Mechanical*. Wiley, New York, 1985.
- [64] Ingo Wegener. *The Complexity of Boolean Functions*. Wiley, New York, 1987.
- [65] William Allen Whitworth. *Choice and Chance*. Hafner Publication Company, New York, 1951.
- [66] H. Woolf, editor. *Webster's New Collegiate Dictionary*. G. and C. Merriam Co., Springfield, Mass., 1973.